Scheduling algorithms to optimize the performance, energy consumption and robustness of HPC applications

Lucas Perotin

THESE

en vue de l'obtention du grade de Docteur, délivré par

l'ECOLE NORMALE SUPERIEURE DE LYON

Ecole Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 29/06/2023, par:

Lucas PEROTIN

Scheduling algorithms to optimize the performance, energy consumption and robustness of HPC applications

Algorithmes d'ordonnancement pour optimiser les performances, la consommation d'énergie et la robustesse des applications de calcul à haute performance (HPC)

Devant le jury composé de :

Fanny Pascual,	Maîtresse de conférences,	Laboratoire d'Informatique de Paris 6,	Rapporteure
Ümit Çatalyürek,	Professeur,	Georgia Institute of Technology,	Rapporteur
Gabriel Antoniu,	Directeur de recherche,	INRIA Rennes Bretagne-Atlantique,	Examinateur
Denis Trystram,	Professeur,	Laboratoire d'Informatique de Grenoble,	Examinateur
Anne Benoit,	Maîtresse de conférences,	ENS de Lyon,	Directrice de thèse
Yves Robert,	Professeur,	ENS de Lyon,	Examinateur

Contents

Résumé

La thèse traite du problème de la résilience dans les systèmes informatiques à grande échelle. En raison du développement rapide de la technologie de calcul à haute performance, il est devenu crucial de développer des mécanismes de tolérance aux pannes efficaces et robustes. Cette recherche se concentre sur l'optimisation des stratégies de sauvegarde, l'analyse de différentes techniques de résilience et le développement de nouvelles approches d'ordonnancement pour traiter les pannes.

Les contributions principales de la thèse incluent l'exploration de l'impact de différents modèles de panne sur les stratégies de sauvegardes, la conception d'algorithmes d'ordonnancement adaptatifs prenant en compte la variabilité des resources, et l'optimisation des stratégies de partage de bande passante d'entrées/sorties (E/S) pour les applications simultanées. Cette thèse comprend également la conception de stratégies de sauvegardes pour les graphes de tâches parallèles.

Dans l'ensemble, cette thèse fournit une analyse approfondie de la résilience et des stratégies de sauvegardes dans les systèmes informatiques de haute performance. Les contributions de cette recherche ont des implications qui concernent l'optimisation des systèmes de calcul scientifique de grande échelle, en répondant à certains défis posés par les erreurs et les pannes dans les processus informatiques.

Introduction en Français

Au cours des dernières années, l'informatique à haute performance (HPC) est devenue essentielle dans de nombreux domaines scientifiques tels que la physique, la chimie et la biologie. Les avancées scientifiques dans ces domaines impliquent souvent des simulations et des calculs complexes, comme la modélisation des champs magnétiques, la compréhension des interactions moléculaires ou la simulation de la dynamique des fluides. De telles opérations demandent une immense puissance de calcul, dépassant souvent les limites des systèmes informatiques actuels et nécessitant une innovation constante pour répondre aux exigences en constante évolution. Comme les méthodes traditionnelles d'amélioration des capacités de calcul, telles que l'augmentation du nombre de transistors sur une seule puce, ont atteint leurs limites physiques, l'accent a été mis sur l'amélioration de la puissance de calcul en augmentant le nombre de composants au sein d'un système. Cette approche permet une exécution parallèle des tâches, répartissant efficacement la charge de calcul sur de nombreux processeurs ou cœurs. Cependant, à mesure que l'échelle de ces plateformes de calcul continue de croître, le défi de maintenir la résilience du système augmente également. Cela devient de plus en plus important car le nombre de composants dans un système HPC est directement proportionnel à la probabilité de défaillances, d'erreurs silencieuses et d'erreurs d'arrêt, qui peuvent finalement compromettre l'exactitude et l'efficacité des calculs scientifiques. En effet, les supercalculateurs de pointe tels que Frontier, Fugaku et LUMI, qui occupent respectivement les 1ère, 2ème et 3ème places dans la liste TOP500 [157], intègrent maintenant des millions de cœurs, avec Sunway TaihuLight (7e) atteignant un pic de 10,6 millions. Ces vastes systèmes de calcul rencontrent souvent des défaillances ou des erreurs d'arrêt, telles que des pannes matérielles ou des blocages dus à des interruptions logicielles anormales. Bien que la probabilité de défaillance pour chaque cœur soit faible, la probabilité globale de défaillance du système est considérablement plus élevée. Par exemple, si le temps moyen entre les pannes (MTBF) pour une seule ressource de calcul est d'environ dix ans, indiquant que la ressource ne devrait connaître une erreur que tous les dix ans en moyenne, en utilisant un million de ressources, le MTBF tombe à cinq minutes, alors que les codes s'exécutant sur de telles plateformes à grande échelle durent généralement des heures ou des jours. À mesure que le besoin de puissance de calcul augmente, les défaillances ne peuvent plus être ignorées, car même une seule erreur peut compromettre la validité des résultats scientifiques, conduisant à des conclusions incorrectes et à une utilisation inefficace des ressources. Ainsi, des mécanismes de tolérance aux pannes doivent être mis en œuvre pour assurer la réussite des applications et la fiabilité des résultats.

Les techniques de résilience visent non seulement à garantir l'exécution correcte des applications scientifiques, mais aussi à minimiser la dégradation des performances et l'utilisation des ressources causées par les défaillances. Elles peuvent être largement classées en approches proactives et réactives. Les approches proactives visent à prédire et à prévenir les défaillances avant qu'elles ne se produisent, tandis que les approches réactives se concentrent sur les mécanismes de récupération et d'adaptation pour faire face aux défaillances après qu'elles se soient produites. De plus, la résilience des applications

scientifiques dépend de divers facteurs tels que la nature de l'application, l'infrastructure matérielle et logicielle sous-jacente et les mécanismes spécifiques de tolérance aux pannes utilisés. Comprendre l'interaction entre ces facteurs est essentiel pour concevoir des stratégies de résilience efficaces et pour évaluer leur impact sur les performances globales du système, et nous étudierons différentes solutions pour bon nombre de ces différents facteurs.

Plus précisément, dans cette thèse, nous distinguons deux types d'erreurs. Les erreurs silencieuses sont un type d'erreur qui se produit pendant l'exécution d'une application et qui n'est pas détecté par le système. Elles peuvent résulter de diverses sources, telles que des inversions de bits dans la mémoire ou des unités arithmétiques et logiques (UAL) défectueuses, et peuvent même être causées par des facteurs externes tels que les rayonnements cosmiques. Le principal défi avec les erreurs silencieuses est de les détecter, car elles n'altèrent que les données ou la sortie d'un algorithme. Des mécanismes de vérification sont disponibles et doivent être utilisés avec prudence pour détecter (et, si possible, corriger) ces erreurs. Contrairement aux erreurs silencieuses, les erreurs d'arrêt sont automatiquement détectées car elles entraînent l'arrêt complet d'une application. Elles peuvent provenir soit d'un composant défectueux, soit d'un bogue dans le code de l'application qui provoque une erreur de segmentation, par exemple. Bien que les erreurs d'arrêt soient plus simples à détecter que les erreurs silencieuses, elles interrompent la progression de l'application, entraînent la perte de toutes les opérations depuis le dernier point de contrôle (s'il y en a eu), et ne peuvent pas être corrigées.

Une approche traditionnelle pour gérer les erreurs d'arrêt dans les systèmes informatiques à grande échelle implique des mécanismes de sauvegardes/réexécutions. Périodiquement, une sauvegarde de l'application est créé, ce qui signifie que l'état de l'application (généralement sa totalité en mémoire) est enregistré dans un stockage fiable. Si une ressource de calcul connaît une défaillance, l'application se met en pause et redémarre à partir du dernier point de contrôle valide. De nombreuses études ont examiné la période de point de contrôle optimale, qui est le temps entre deux points de contrôle consécutifs, de manière à minimiser le gaspillage. Si les points de contrôle sont pris trop fréquemment, un temps précieux est consacré aux opérations d'entrées/sorties (E/S). À l'inverse, si les points de contrôle sont trop rares, du temps est perdu à retraiter de grandes parties de l'application après chaque erreur (voir Figure 1.1). De manière intéressante, la fiabilité était déjà une préoccupation aux premiers jours de l'informatique : dans les années 1970, Young a proposé une approximation initiale du temps optimal entre deux points de contrôle qui minimise la durée de calcul totale [173]. Daly a ensuite affiné l'approximation de Young [47], et la période de point de contrôle optimale est (approximativement) donnée par la formule de Young/Daly comme $W_{YD} = \sqrt{2\mu C}$, où μ est le temps moyen entre les pannes de l'application et C est la durée de la sauvegarde. Cette formule s'applique aux applications où un point de contrôle peut être pris à tout moment pendant le calcul, ce qui est le cas pour les applications à charge divisible [27, 136].

Pour relever le défi de la résilience dans les systèmes HPC, cette thèse est organisée en deux parties principales, chacune comprenant trois chapitres qui se concentrent sur différents aspects de la résilience dans le contexte des HPC. La première partie approfondit la résilience sans point de contrôle, en mettant principalement l'accent sur la planification des tâches, la détection et la récupération des erreurs, tandis que la deuxième partie étudie différentes stratégies de point de contrôle et leur efficacité pour maintenir la résilience dans différents scénarios.

Plus précisément, le chapitre 2 se concentre sur la planification résiliente des tâches parallèles malléables sur des plateformes de HPC. Les tâches malléables offrent la flexibilité

de choisir l'allocation de processeurs avant l'exécution et peuvent adhérer à divers modèles de gain de vitesse, qui représentent le temps d'exécution d'une tâche en fonction du nombre de cœurs qui lui sont alloués. L'objectif est de minimiser le temps total d'achèvement, tout en tenant compte de la possibilité d'échecs de tâches dus à des erreurs silencieuses, qui peuvent nécessiter une nouvelle exécution. Ce chapitre généralise le cadre de planification classique pour les tâches sans défaillance et introduit deux algorithmes de planification résilients, présente de nouveaux ratios d'approximation et démontre leur efficacité au moyen d'un ensemble étendu de simulations.

Le chapitre 3 étend l'étude à la planification en ligne des graphes de tâches malléables sur des systèmes multiprocesseurs, où les tâches ne sont découvertes qu'à la fin de leurs prédécesseurs. L'objectif est de minimiser le temps total d'achèvement selon divers modèles réalistes de gain de vitesse. Nous concevons un nouvel algorithme en ligne et dérivons des ratios de compétitivité constants pour ces modèles de gain de vitesse. Nous établissons également des bornes inférieures sur la compétitivité de tout algorithme dont l'allocation de processeur ne dépend que des paramètres de la tâche et pas de sa position dans le graphe, et nous montrons que notre algorithme a le meilleur ratio de compétitivité absolu pour cette classe sous ce modèle.

Dans le chapitre 4, le focus se déplace vers la planification consciente des risques de tâches indépendantes sur une plateforme avec une capacité de ressources variable. La dépendance croissante des sources d'énergie renouvelable, telles que l'énergie solaire et éolienne, a entraîné des variations dans le coût, la disponibilité et l'intensité en carbone de l'énergie. Cela nécessite le développement d'algorithmes de planification qui peuvent s'adapter à ces fluctuations. L'objectif d'optimisation de ce chapitre est le débit, défini comme la fraction de temps consacrée à des calculs efficaces, excluant la ré-exécution. Nous introduisons plusieurs algorithmes innovants qui: (i) déterminent la fraction de ressources sûres à utiliser; (ii) maintiennent un indice de risque pour chaque machine; et (iii) atteignent un équilibre de charge global tout en affectant les tâches plus longues aux machines plus sûres. Les performances de ces algorithmes sont évaluées à l'aide de traces de flux de travail réelles ainsi que de traces synthétiques, résultant en une augmentation moyenne du débit sans compromettre d'autres métriques comme l'étirement maximal ou moyen.

La deuxième partie traite de l'impact des erreurs de type fail-stop. Le chapitre 5 étudie les stratégies de sauvegardes pour les tâches parallèles soumises aux erreurs de type fail-stop. Alors que la stratégie optimale est bien établie lorsque les temps entre deux erreurs suivent une distribution exponentielle, elle est inconnue pour les distributions de temps non exponentielles. Nous abordons les idées fausses dans la littérature récente et proposons une stratégie générale qui maximise l'efficacité attendue jusqu'à la prochaine erreur. Nous démontrons que cette stratégie est asymptotiquement optimale pour les très longues tâches. À travers des simulations approfondies, la nouvelle stratégie est montrée pour surpasser constamment la stratégie classique de Young/Daly pour diverses distributions d'erreur. Dans certains cas, elle réduit le temps d'exécution d'un facteur de 1.9 en moyenne, et jusqu'à un facteur de 4.2 pour les plateformes récemment déployées.

Dans le chapitre 6, le focus se déplace vers les stratégies de sauvegardes pour les processus comprenant plusieurs tâches s'exécutant sur des plateformes parallèles. L'objectif est de minimiser le temps d'exécution total attendu. Alors que la formule de Young/Daly fournit la période de sauvegarde optimale pour une tâche unique, la situation devient plus complexe avec plusieurs tâches concurrentes. Nous explorons si l'ajout de sauvegardes supplémentaires à chaque tâche au-delà de la stratégie de Young/Daly est bénéfique dans un contexte global et présentons des résultats théoriques négatifs pour conserver la période

de Young/Daly lorsque de nombreuses tâches sont exécutées simultanément. Enfin, nous concevons de nouvelles stratégies de sauvegarde qui garantissent une exécution efficace avec une forte probabilité. Des expériences approfondies démontrent la nécessité d'aller au-delà de la période de Young/Daly et de faire des sauvegardes plus souvent dans diverses applications/plateformes.

Enfin, parce que la vérification nécessite de grandes opérations d'entrées/sorties (E/S) pour stocker et lire les données, elle augmente considérablement la charge sur le système d'E/S. Plusieurs applications qui tentent de vérifier (ou de récupérer) simultanément devront partager la bande passante d'E/S. Le chapitre 7 revoit les stratégies de partage de bande passante d'E/S pour les applications HPC. Il compare les approches bien connues telles que la sérialisation des opérations (FCFS) et le partage équitable de la bande passante entre les opérations d'E/S concurrentes (MinYield) avec des approches plus récentes telles que Set-10, qui attribue des priorités aux applications en fonction de la durée moyenne de leurs itérations. Nous introduisons plusieurs nouvelles stratégies de partage de bande passante, y compris des algorithmes gloutons simples et plus complexes, et évaluons leurs performances par rapport aux stratégies existantes. Les stratégies proposées ne dépendent pas de la connaissance préalable du comportement de l'application, tel que la durée des phases de travail, le volume d'opérations d'E/S ou la périodicité. Nous présentons un cadre rigoureux, appelé *fenêtres à l'état stable*, pour dériver des bornes sur le rapport de compétition de toutes les stratégies de partage de bande passante pour trois objectifs différents : le rendement minimal, l'utilisation de la plateforme et l'efficacité globale. Cette évaluation théorique est complétée par des simulations approfondies à l'aide de traces synthétiques et réalistes.

Chacun des six chapitres de contributions de cette thèse correspond à une publication scientifique dont je suis co-auteur. Dans l'ensemble, cette thèse vise à fournir une analyse approfondie de la résilience et de la vérification dans les systèmes HPC. En abordant différents aspects de la résilience, notamment la planification, les stratégies de vérification et les techniques de partage de bande passante d'E/S, cette thèse apporte de nouvelles contributions à la compréhension et à l'optimisation des systèmes HPC. De plus, mon travail a visé à améliorer la fiabilité et l'efficacité du calcul scientifique dans une gamme diversifiée d'applications, aidant les chercheurs et les praticiens à surmonter les défis posés par les pannes et les erreurs dans leur travail de calcul.

Ma liste complète de publications et de soumissions est fournie dans le Chapitre 9. Par souci de concision, les travaux réalisés dans la publication [C2] et la soumission [S2] ne sont pas inclus dans cette thèse. Ces travaux étendent les problèmes d'ordonnancement classiques avec plusieurs ressources (comme les processeurs, le cache, la mémoire, les E/S ou les ressources réseau). Nous introduisons un nouvel algorithme d'ordonnancement multi-ressources avec des garanties théoriques, et nous confirmons expérimentalement ses bonnes performances. Cet algorithme est complexe et nécessite plusieurs bijections pour transformer l'instance ainsi que de la programmation linéaire, et le code est disponible publiquement à l'adresse `https://gitlab.inria.fr/luperoti/mrsa`. Dans cette thèse, nous omettons également les travaux réalisés dans la publication [C3], étant donné qu'un nouvel algorithme et une analyse plus précise, soumis dans [S5], rendent les résultats précédents obsolètes.

Chapter 1

Introduction

In recent years, high-performance computing (HPC) has become essential in a wide range of scientific domains, such as physics, chemistry, and biology. Scientific breakthroughs in these domains often involve complex simulations and calculations, like modeling magnetic fields, understanding molecular interactions, or simulating fluid dynamics. Such computations demand immense computational power, frequently pushing the limits of current computing systems and necessitating constant innovation to keep up with the ever-growing requirements. As traditional methods of enhancing computing capabilities, like increasing the number of transistors on a single chip, have reached physical limitations, the focus has shifted toward improving computing power by augmenting the number of components within a system. This approach enables parallel execution of tasks, effectively distributing the computational workload across numerous processors or cores. However, as the scale of these computing platforms continues to grow, so does the challenge of maintaining system resilience. This becomes increasingly important as the number of components in an HPC system is directly proportional to the likelihood of faults, silent errors, and fail-stop errors, which can ultimately compromise the accuracy and efficiency of scientific computations. Indeed, cutting-edge supercomputers like Frontier, Fugaku, and LUMI, which rank 1st, 2nd, and 3rd in the TOP500 list [157], respectively, now incorporate millions of cores, with Sunway TaihuLight (7th) reaching a peak of 10.6 million. These vast computing systems often encounter failures or fail-stop errors, such as hardware malfunctions or crashes. Although the probability of failure for each core is low, the overall system's failure probability is considerably higher. For example, if the Mean Time Between Failure (MTBF) for a single computing resource is roughly ten years, indicating that the resource should experience an error only every ten years on average, using one million resources, the MTBF drops to five minutes, while codes running on such extreme-scale platforms usually last for hours or days. As the need for computing power grows, failures can no longer be overlooked, as even a single error can compromise the validity of scientific results, leading to incorrect conclusions and wasted resources. Thus, fault-tolerance mechanisms must be implemented to ensure the successful completion of applications and the reliability of results.

Resilience techniques not only aim at ensuring the correct execution of scientific applications but also at minimizing the performance degradation and resource utilization caused by failures. They can be broadly classified into proactive and reactive approaches. Proactive approaches strive for predicting and preventing failures before they occur, while reactive approaches focus on recovery and adaptation mechanisms to deal with failures after they have happened. Moreover, the resilience of scientific applications depends on various factors, such as the nature of the application, the underlying hardware and software

infrastructure, and the specific fault-tolerance mechanisms employed. Understanding the interplay between these factors is essential for designing effective resilience strategies and for evaluating their impact on the global system performance, and we will study different solutions for many of these various factors.

More precisely, in this thesis, we distinguish two types of errors. Silent errors are a type of errors that occur during an application's execution and go undetected by the system. They can result from various sources, such as bit-flips in memory or malfunctioning Arithmetic Logic Units (ALUs), and may even be caused by external factors like cosmic radiation. The primary challenge with silent errors is to detect them, since they only alter the data or output of an algorithm. Verification mechanisms are available and must be used carefully to detect (and, if possible, correct) these errors. In contrast to silent errors, fail-stop errors are automatically detected since they lead to a complete halt of an application. They can arise from either a non-functioning component or a bug in the application code that causes a segmentation fault, for example. Although fail-stop errors are simpler to detect than silent errors, they interrupt application progress, cause the loss of all computations since the last checkpoint (if any), and cannot be corrected.

A traditional approach for handling fail-stop errors in extreme-scale computing systems involves checkpoint/rollback mechanisms. Periodically, an application checkpoint is created, meaning the application's state (typically its entire memory footprint) is saved to reliable storage. If any computing resource experiences a failure, the application pauses and restarts from the last valid checkpoint. Many studies have investigated the optimal checkpointing period, which is the time between two consecutive checkpoints, so that the waste is minimized. If checkpoints are taken too frequently, some valuable time is spent on I/O operations. Conversely, if checkpoints are too sparse, time is wasted reprocessing large parts of the application after each failure (see Figure 1.1). Interestingly, reliability was already a concern in the early days of computing: in the 1970s, Young proposed an initial approximation of the optimal time between two checkpoints that minimizes the total computation duration [173]. Daly later refined Young's approximation [47], and the optimal checkpointing period is (approximately) given by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$, where μ is the application MTBF (Mean Time Between Failures) and C is the checkpoint duration. This formula applies to applications where a checkpoint can be taken anytime during the computation, which is the case for divisible-load applications [27, 136].

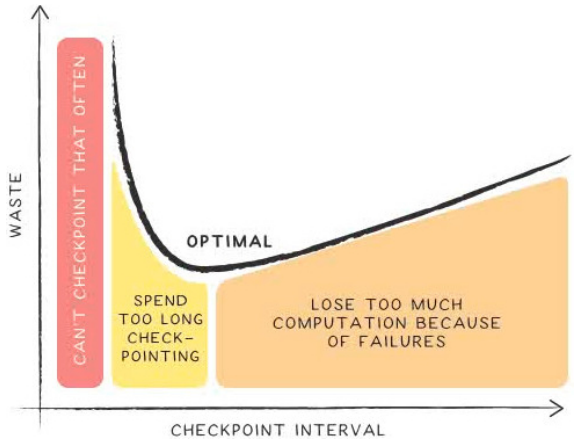


Figure 1.1: Trade-off for the optimal checkpoint period.

To address the challenge of resilience in HPC systems, this thesis is organized into two main parts, each comprising three chapters that focus on different aspects of resilience in the context of HPC. The first part delves into resilience without checkpointing, emphasizing mostly on task scheduling, error detection, and recovery, while the second part investigates various checkpointing strategies and their effectiveness in maintaining resilience under different scenarios.

More specifically, Chapter 2 focuses on the resilient scheduling of moldable parallel jobs on HPC platforms. Moldable jobs provide the flexibility of choosing a processor allocation before execution and can adhere to various speedup models, which is the function representing the execution time of a job given the number of cores allocated to it. The objective is to minimize the overall completion time, while accounting for the possibility of job failures due to silent errors, which may necessitate re-execution. This chapter generalizes the classical scheduling framework for failure-free jobs and introduces two resilient scheduling algorithms, presents new approximation ratios, and demonstrates their effectiveness through an extensive set of simulations.

Chapter 3 extends the study to the online scheduling of moldable task graphs on multiprocessor systems, where tasks are discovered only upon the completion of their predecessors. The goal is to minimize the overall completion time under various realistic speedup models. We design a novel online algorithm and derive constant competitive ratios for these speedup models. We also establish lower bounds on the competitiveness of any algorithm whose processor allocation depends only on task parameters and not on its position in the graph, and we show that our algorithm has the absolute best competitive ratio for this class under this model.

In Chapter 4, the focus shifts to the risk-aware scheduling of independent jobs on a platform with variable resource capacity. The increasing reliance on renewable energy sources, such as solar and wind power, has led to variations in the cost, availability, and carbon intensity of power. This necessitates the development of scheduling algorithms that can adapt to these fluctuations. The optimization objective in this chapter is the goodput, defined as the fraction of time devoted to effective computations, excluding re-execution. We introduce several innovative algorithms that: (i) determine the safe fraction of resources to use; (ii) maintain a risk index for each machine; and (iii) achieve global load balance while mapping longer jobs to safer machines. The performance of these algorithms is assessed using actual workflow traces as well as synthetic traces, resulting in an average increase in goodput without compromising other metrics like maximum or average stretch.

The second part deals with the impact of fail-stop errors. Chapter 5 studies checkpointing strategies for parallel jobs subject to fail-stop errors. While the optimal strategy is well-established when failure inter-arrival times follow an Exponential distribution, it remains unknown for non-memoryless failure distributions. We address misconceptions in recent literature and propose a general strategy that maximizes the expected efficiency until the next failure. We demonstrate that this strategy is asymptotically optimal for very long jobs. Through extensive simulations, the new strategy is shown to consistently outperform the classic Young/Daly strategy for various failure distributions. In some cases, it reduces execution time by a factor of 1.9 on average and up to a factor of 4.2 for recently deployed platforms.

In Chapter 6, the focus shifts to checkpointing strategies for workflows comprising multiple tasks executing on parallel platforms. The objective is to minimize the expected total execution time. While the Young/Daly formula provides the optimal checkpointing

period for a single task, the situation becomes more complex with multiple concurrent tasks. We explore whether adding extra checkpoints to each task beyond the Young/Daly strategy is beneficial in a global context and present theoretical negative results for retaining the Young/Daly period when many tasks execute concurrently. Finally, we design novel checkpointing strategies that guarantee efficient execution with high probability. Comprehensive experiments demonstrate the need to go beyond the Young/Daly period and to checkpoint more often in various application/platform settings.

Finally, because checkpointing requires large I/O operations to store and read the data, it dramatically increases the overhead on the I/O system. Several applications that try and checkpoint (or recover) simultaneously will need to share the I/O bandwidth. Chapter 7 revisits I/O bandwidth-sharing strategies for HPC applications. It compares well-known approaches such as serializing operations (FCFS) and fair-sharing bandwidth across concurrent I/O operations (MinYield) with newer approaches like Set-10, which assigns priorities to applications based on the average length of their iterations. We introduce several new bandwidth-sharing strategies, including simple greedy algorithms and more complex ones, and assess their performance against existing strategies. The proposed strategies do not rely on prior knowledge of application behavior, such as work phase lengths, I/O operation volume, or periodicity. We present a rigorous framework, called *steady-state windows*, to derive bounds on the competitive ratio of all bandwidth-sharing strategies for three different objectives: minimum yield, platform utilization, and global efficiency. This theoretical assessment is complemented by extensive simulations using synthetic and realistic traces.

Each of the six chapters of contributions in this thesis corresponds to a scientific publication that I have co-authored. Altogether, this thesis aims at providing a thorough analysis of resilience and checkpointing in HPC systems. By addressing various aspects of resilience, including scheduling, checkpointing strategies, and I/O bandwidth-sharing techniques, this thesis makes new contributions to the understanding and optimization of HPC systems. Furthermore, my work aspires to enhance the reliability and efficiency of scientific computing in a diverse range of applications, helping researchers and practitioners overcome the challenges posed by faults and errors in their computational work.

My complete list of publications and submissions is provided in Chapter 9. For the sake of conciseness, the work done in Publication [C2] and Submission [S2] are not included in this thesis. This work involves extending classical scheduling problems with multiple resources (such as computing cores, cache, memory, I/O or network resources). We design a new multi-resource scheduling algorithm with theoretical guaranties, and we experimentally confirm its good performance. This algorithm is complex and requires multiple bijections to transform the instance, as well as some linear programming, and the code is publicly available at `https://gitlab.inria.fr/luperoti/mrsa`. In this thesis, we also omit the work done in Publication [C3], since a newer algorithm and sharper analysis, submitted in [S5], makes the previous results obsolete.

Part I

Robust scheduling of parallel jobs

Chapter 2

Resilient Scheduling of Moldable Parallel Jobs to Cope with Silent Errors

We study the resilient scheduling of moldable parallel jobs on high-performance computing (HPC) platforms. Moldable jobs allow for choosing a processor allocation before execution, and their execution time obeys various speedup models. The objective is to minimize the overall completion time of the jobs, or the makespan, when jobs can fail due to silent errors and hence may need to be re-executed after each failure until successful completion. Our work generalizes the classical scheduling framework for failure-free jobs. To cope with silent errors, we introduce two resilient scheduling algorithms, LPA-LIST and BATCH-LIST, both of which use the LIST strategy to schedule the jobs. Without knowing a priori how many times each job will fail, LPA-LIST relies on a local strategy to allocate processors to the jobs, while BATCH-LIST schedules the jobs in batches and allows only a restricted number of failures per job in each batch. We prove new approximation ratios for the two algorithms under several prominent speedup models (e.g., roofline, communication, Amdahl, power, monotonic, and a mixed model). An extensive set of simulations is conducted to evaluate different variants of the two algorithms, and the results show that they consistently outperform some baseline heuristics. Overall, our best algorithm is within a factor of 1.6 of a lower bound on average over the entire set of experiments, and within a factor of 4.2 in the worst case. This chapter mostly corresponds to Publication [J1], which was itself an extension from Publication [C1] (see Chapter 9). However, it also contains some results obtained later, such as the asymptotic optimality of BATCH-LIST, which revisits results from Publication [C3]. Finally, this chapter also incorporates results from Submission [S5], which revisited the study of the speedup models.

2.1 Introduction

Scheduling parallel jobs on high-performance computing (HPC) platforms is crucial for improving the application and system performance. In the scheduling literature, a *moldable* job is a parallel job that can be executed on an arbitrary but fixed number of processors, with an execution time depending on the number of processors on which it is executed. More precisely, a moldable job allows a variable set of resources for scheduling but requires a fixed set of resources to execute, which the job scheduler must allocate before it starts the job. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (*rigid* jobs) and to a variable dynamic allocation (*malleable* jobs) [59]. Moldable jobs can easily adapt to the amount of available resources, contrarily to rigid jobs, while being easy to design and implement, contrarily to malleable jobs. Thus, many

n	Number of tasks
P	Total number of processors
λ	Error rate per unit of work (failures follow exponential law)
$\mathbf{f}$	Vector containing the number of failures for each task
$\vec{p}$	Vector containing the number of processors for each iteration of a given task
$\mathbf{p}$	Vector containing the $\vec{p}$ for all tasks
$\vec{s}$	Vector containing the starting time of each iteration of a given task
$\mathbf{s}$	Vector containing the $\vec{s}$ for all tasks
q_j	Probability of failure for a given task
w_j	Work of a given task (sequential execution time)
d_j	Inherently sequential fraction of a task (cannot be parallelized)
c_j	Communication overhead per processor for a task
$\bar{p}_j$	Maximum degree of parallelism of a task
$t_j(p_j)$	Execution time of task j when allocated p_j processors
Roo	Roofline Model
Com	Communication Model
Amd	Amdahl Model
Mix	Mix Model

Table I: Summary of main notations for Chapter 2.

computational kernels in scientific libraries are provided as moldable jobs that can be deployed on a wide range of processor numbers.

Because of the importance and wide availability of moldable jobs, scheduling algorithms for such jobs have been extensively studied. An important objective is to minimize the overall completion time, or makespan, for a set of jobs that are either all known before execution (offline setting) or released on-the-fly (online setting). Many prior works have published approximation algorithms or inapproximability results for both settings. These results notably depend upon the speedup model of the jobs. Indeed, consider a job whose execution time is $t(p)$ with p processors ($1 \leq p \leq P$, and P denotes the total number of processors on the platform). An arbitrary speedup model allows $t(p)$ to take any value, but realistic models call for $t(p)$ non-increasing with p: after all, if $t(p+1) > t(p)$, then why use that extra processor? Several speedup models have been introduced and analyzed, including the roofline model, the communication model, the Amdahl's model, the power model, and the (more general) monotonic model, where the area of the job $p \cdot t(p)$ is non-decreasing with p. Section 2.2 presents a survey of some important results for all these models.

In this chapter, we revisit the problem of scheduling moldable jobs in a resilience framework. Unlike the classical problem without job failures, we consider *failure-prone jobs* that may need to be re-executed several times before successful completion. This is primarily motivated by the threat of silent errors (a.k.a. *silent data corruptions or SDCs*), which strike large-scale high-performance computing (HPC) platforms at a rate proportional to the number of floating-point (CPU) operations and/or the memory footprint of the applications (bit flips) [128,178]. When a silent error strikes, even though any bit can be corrupted, the execution continues (unlike fail-stop errors), hence the error is transient, but it may dramatically impact the result of a running application. Coping with silent errors is a major challenge on today's HPC platforms [119] and it will become even more important at exascale [80]. Fortunately, many silent errors can be accurately detected by verifying the integrity of data using dedicated, lightweight detectors (e.g., [41, 43, 74, 166]).

When considering job failures caused by silent errors, we assume the availability of ad-hoc detectors.

To model this resilient scheduling problem, we focus on a general setting, where the aim is to schedule a set of moldable jobs subject to a failure scenario that specifies the number of failures for each job before successful completion. The failure scenario is, however, not known a priori, but only discovered as failed executions manifest themselves when the jobs complete. Hence, the scheduling decisions must be made *dynamically* on-the-fly: whenever an error has been detected, the job must be re-executed. As a result, even for the same set of jobs, different schedules may be produced, depending on the failure scenario that occurred in a particular execution. Intuitively, the problem lies in between an offline problem (where all the jobs are known before the execution starts) and an online problem (where the jobs are revealed on-the-fly). The goal is to minimize the makespan for any set of jobs under any failure scenario. Since the problem is clearly NP-complete (as it generalizes the NP-complete failure-free scheduling problem), we aim at designing approximation algorithms that guarantee a makespan within a provable factor of the optimal makespan, independently of the jobs' failure scenarios.

Extending the literature on scheduling moldable jobs in the failure-free setting, this work lays the theoretical and practical foundation for scheduling such jobs on failure-prone platforms. Our key contributions are the design and analysis of two resilient scheduling algorithms with new approximation results for various speedup models. We further show that the two algorithms achieve good practical performance using an extensive set of simulations. The following summarizes our main results:

- We present a formal model for the problem of resilient scheduling of moldable jobs on failure-prone platforms. The model formulates both the worst-case and average-case performance of an algorithm for general speedup models and under arbitrary failure scenarios.
- We design a resilient scheduling algorithm, called LPA-LIST, that relies on a local processor allocation strategy and list scheduling to achieve $O(1)$-approximation for some prominent speedup models, including the roofline model, the communication model, the Amdahl's model, and a mixed model. For the communication model, our approximation ratio improves on that of the literature for failure-free jobs. We also show that the algorithm is $\Theta(P^{1/4})$-approximation for the power model and $\Theta(P^{1/2})$-approximation for the general monotonic model. All of these results apply to both worst-case and average-case performance.
- We design another resilient scheduling algorithm, called BATCH-LIST, which schedules the jobs in batches using the list strategy, and each job is allowed only a restricted number of failures per batch. We prove a tight $\Theta(\log_2 f_{\max})$-approximation for the algorithm under arbitrary speedup model in the worst case, where $f_{\max}$ is the maximum number of failures of any job in a failure scenario. We also prove an $\omega(1)$ lower bound on the average-case performance of the algorithm.
- We derive a lower bound on the competitiveness of any algorithm under the arbitrary speedup model, and show that no algorithm may have a competitive ratio in $o(\log_2 f_{\max})$, showing the near-optimality of BATCH-LIST in the worst case.
- We conduct an extensive set of simulations to evaluate and compare different variants of the two algorithms. The results show that they consistently outperform some baseline heuristics. In particular, the first algorithm (LPA-LIST) performs better for the roofline and communication models, while the second algorithm (BATCH-LIST) performs better for the other models. Overall, our best algorithm is within a factor of 1.6 of a lower bound on average and within a factor of 4.2 in the worst case for all

speedup models.

The rest of this chapter is organized as follows. Section 2.2 surveys related work. The formal model and problem statement are presented in Section 2.3. In Section 2.4, we describe the two main algorithms and analyze their performance, providing several new approximation results. Section 2.5 is devoted to proving a lower bound of any deterministic online algorithm for the arbitrary speedup model. Section 2.6 presents an extensive set of simulation results and highlights the main findings. Finally, Section 2.7 concludes the chapter and discusses future directions.

2.2 Related Work

In this section, we review some related work on scheduling moldable jobs without failures, and we highlight the differences of these models with the one studied in this chapter.

2.2.1 Offline Scheduling of Independent Moldable Jobs

In offline scheduling, all jobs are known a priori along with each job's execution time $t(p)$ as a function of the processor allocation p. The following reviews some results in the failure-free setting under various job speedup models (the definitions of these models can be found in Section 2.3.1).

Roofline Model: This model assumes linear speedup up to a bounded degree of parallelism $\bar{p}$. Some authors have considered this model for moldable jobs with precedence constraints. We are not aware of any results for independent moldable jobs. In this chapter, we show that allocating exactly $\bar{p}$ processors to the job and then scheduling all jobs greedily gives a 2-approximation when jobs are subject to failures.

Communication Model: This model assumes a communication overhead when using more than one processor. Havill and Mao [78] presented a shortest execution time (SET) algorithm, which selects a number of processors that minimizes the job's execution time (they use around $\sqrt{w/c}$ processors when $t(p) = w/p + (p-1)c$), and schedules each job as early as possible. They showed that SET has an approximation ratio around 4. In this chapter, we present an improved algorithm with an approximation ratio of 3. Furthermore, the algorithm is able to handle job failures. Dutton and Mao [54] presented an earliest completion time (ECT) algorithm, which allocates processors for each job that minimizes its completion time based on the current schedule. They proved tight approximation ratios of ECT for $P \leq 4$ processors and presented a general lower bound of 2.3 for arbitrary P. Kell and Havill [102] presented algorithms with improved approximation ratios for $P \leq 3$ processors.

Monotonic Model: This model assumes that the execution time is a non-increasing function and the area (product of processor allocation and execution time) is a non-decreasing function of the processor allocation. Examples of this model include Amdahl's speedup [3], i.e., $t(p) = w(\frac{1-d}{p} + d)$ with $d \in [0, 1]$, and the power speedup $t(p) = w/p^{\delta}$ [73, 148] with $\delta \in [0, 1]$. Belkhale and Banerjee [16] presented a $2/(1 + 1/P)$-approximation algorithm by starting from a sequential LPT schedule and then iteratively incrementing the processor allocations. Błażewicz et al. [29] presented a 2-approximation algorithm while relying on an optimal continuous schedule, in which the processor allocation of a job may not be integral. Mounié et al. [123] presented a $(\sqrt{3} + \epsilon)$-approximation algorithm using a two-phase approach and dual approximation. Using the same techniques, they later improved the approximation ratio to $1.5 + \epsilon$ [124]. Jansen and Land [90] showed the same $1.5 + \epsilon$ ratio but with a lower runtime complexity, when the execution time functions

of the jobs admit certain compact encodings. They also proposed a PTAS for the problem.

Arbitrary Model: In this model, the execution time $t(p)$ is an unrestricted function of the processor allocation p. This model can be reduced to the monotonic model by scanning all possible allocations and discarding those with both larger execution time and area. Turek et al. [159] presented a 2-approximation list-based algorithm and a 3-approximation shelf-based algorithm. Ludwig and Tiwari [116] improved the 2-approximation result with lower runtime complexity. When each job only admits a subset of all possible processor allocations, Jansen [89] presented a $(1.5+\epsilon)$-approximation algorithm, which is the strongest result possible for any polynomial-time algorithm, since the problem does not admit an approximation ratio better than 1.5 unless $\mathcal{P} = \mathcal{NP}$ [96]. However, when the number of processors is a constant or polynomially bounded by the number of jobs, Jansen et al. [91, 92] showed that a PTAS exists.

2.2.2 Online Scheduling of Independent Moldable Jobs

In online scheduling, jobs are released one by one to the scheduler, and each released job must be scheduled irrevocably before the next job is revealed. As some algorithms discussed in the previous section (e.g., [54, 78, 102]) make scheduling decisions independently for each job, their results can be directly applied to this online problem with the corresponding competitive ratios. In contrast, other algorithms rely on the information about all jobs to make global scheduling decisions, so these algorithms and their approximation results are not directly applicable to the online problem. In this online problem under the arbitrary speedup model, Ye et al. [170] presented a technique to transform any ρ-bounded algorithm[1] for rigid jobs to a 4ρ-competitive algorithm for moldable jobs. Then, relying on a 6.66-bounded algorithm for rigid jobs [86, 171], they gave a 26.65-competitive algorithm for moldable jobs. Both algorithms are based on building shelves. They also provided an improved algorithm with a competitive ratio of 16.74 [170].

The problem studied in this chapter can be considered as semi-online, since all jobs are known to the scheduler offline but their failure scenarios are revealed online. We point out that the transformation technique by Ye et al. [170] does not apply here, since it implicitly assumes the independence of all jobs, whereas the different executions of the same job in our problem (due to failures) have linear dependence.

2.3 Models

In this section, we formally describe the models, and present the resilient scheduling problem.

2.3.1 Job and Speedup Models

We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n parallel jobs to be executed on a platform consisting of P identical processors. All jobs are released at the same time, corresponding to the batch scheduling scenario in an HPC environment. We focus on *moldable* jobs, which can be executed using any number of processors at launch time. The number of processors allocated cannot be changed once a job has started executing. For each job $J_j \in \mathcal{J}$, $t_j(p_j)$

[1] An algorithm for rigid jobs is said to be ρ-bounded if its makespan is at most ρ times the lower bound $L = \max\left(\frac{\sum_j t_j p_j}{P}, \max_j t_j\right)$, where t_j denotes the execution time of job J_j, and p_j denotes its processor allocation.

denotes its execution time when allocated $p_j \in \{1, 2, \ldots, P\}$ processors[2], and the *area* of the job is defined as $a_j(p_j) = p_j \times t_j(p_j)$.

Let w_j denote the total work of job J_j (or its sequential execution time $t_j(1)$). The parallel execution time $t_j(p_j)$ of the job when allocated p_j processors depends on the speedup model. We consider several speedup models:

- *Roofline model*: linear speedup up to a bounded degree of parallelism $\bar{p}_j \in [1, P]$, i.e., $t_j(p_j) = w_j/p_j$ for $p_j \leq \bar{p}_j$, and $t_j(p_j) = w_j/\bar{p}_j$ for $p_j > \bar{p}_j$;
- *Communication model*: there is a communication overhead $c_j \geq 0$ per processor when more than one processor is used, i.e., $t_j(p_j) = w_j/p_j + (p_j - 1)c_j$;
- *Amdahl's model*: this is a particular case of the monotonic model with $t_j(p_j) = w_j(\frac{1-d_j}{p_j} + d_j)$, where $d_j \in [0, 1]$ denotes the inherently sequential fraction of the job;
- *Mix model*: this mixed model combines Roofline, Communication and Amdahl's models with $t_j(p_j) = \frac{w_j(1-d_j)}{\min(p, \bar{p}_j)} + w_j d_j + (p_j - 1)c_j$, which could capture more realistically the speedups of some complex applications;
- *Power model*: this is another particular case of the monotonic model with $t_j(p_j) = w_j/p_j^{\delta_j}$, where $\delta_j \in [0, 1]$ is a constant parameter;
- *Monotonic model*: the execution time (resp. area) is a non-increasing (resp. non-decreasing) function of the number of allocated processors, i.e., $t_j(p_j) \geq t_j(p_j + 1)$ and $a_j(p_j) \leq a_j(p_j + 1)$;
- *Arbitrary model*: there are no constraints on $t_j(p_j)$.

In all of these models, the *speedup* of job J_j with p_j processors is given by $\sigma_j(p_j) = \frac{t_j(1)}{t_j(p_j)}$.

2.3.2 Failure Model

We consider silent errors (or SDCs) that could cause a job to produce erroneous results after an execution attempt. Further, we assume that such errors can be detected using lightweight detectors with negligible overhead at the end of an execution. In that case, the job needs to be re-executed followed by another error detection. This process repeats until the job completes successfully without errors.

Let $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ denote a *failure scenario*, i.e., a vector of the number of failed execution attempts for all jobs, during a particular execution of the job set $\mathcal{J}$. Note that the number of times a job will fail is unknown to the scheduler a priori, and the failure scenario $\mathbf{f}$ becomes known only after all jobs have successfully completed without errors.

2.3.3 Problem Statement

We study the following *resilient scheduling* problem: Given a set of n moldable jobs, find a schedule on P identical processors under any failure scenario $\mathbf{f}$. In this context, a *schedule* is defined by the following two decisions:

- *Processor allocation*: a collection $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)$ of processor allocation vectors for all jobs, where vector $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \ldots, p_j^{(f_j+1)})$ specifies the number of processors allocated to job J_j at different execution attempts until success. Note that processor allocation can change for each new execution attempt of a job.
- *Starting time*: a collection $\mathbf{s} = (\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n)$ of starting time vectors for all jobs, where vector $\vec{s}_j = (s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(f_j+1)})$ specifies the starting times for job J_j at different execution attempts until success.

[2]In this work, we do not allow fractional processor allocation, which could otherwise be realized by timesharing a processor among multiple jobs.

The objective is to minimize the overall completion time of all jobs, or *makespan*, under any failure scenario. Suppose an algorithm makes decisions $\mathbf{p}$ and $\mathbf{s}$ for a job set $\mathcal{J}$ during a failure scenario $\mathbf{f}$. Then, the makespan of the algorithm for this scenario is defined as:

$$T(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) = \max_{1 \le j \le n} \left(s_j^{(f_j+1)} + t_j(p_j^{(f_j+1)}) \right) . \tag{2.1}$$

Both scheduling decisions should be made with the following two constraints: (1) the number of processors used at any time should not exceed the total number P of available processors; (2) a job cannot be re-executed if its previous execution attempt has not yet been completed.

As the problem generalizes the failure-free moldable job scheduling problem, which is known to be $\mathcal{NP}$-complete for $P \ge 5$ processors [53], the resilient scheduling problem is also $\mathcal{NP}$-complete. We therefore consider approximation algorithms. A scheduling algorithm ALG is said to be an *r-approximation*[3] if its makespan is at most r times that of an optimal scheduler for any job set $\mathcal{J}$ under any failure scenario $\mathbf{f}$, i.e.,

$$\sup_{\mathcal{J}, \mathbf{f}} \frac{T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} = r , \tag{2.2}$$

where $T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)$ denotes the makespan produced by an optimal scheduler with scheduling decisions $\mathbf{p}^*$ and $\mathbf{s}^*$.

2.3.4 Worst-Case vs. Average-Case Analysis

The problem above is agnostic of the failure scenario, which is given as an input of the scheduling problem. A scheduling algorithm is an r-approximation only if it achieves a makespan at most r times the optimal for *any possible* failure scenario. This can be viewed as the *worst-case* analysis.

In contrast, some practical settings may call for an *average-case* analysis. In practice, each job $J_j \in \mathcal{J}$ could fail with a probability q_j in each execution attempt, independent of the number of previous failures. For instance, consider silent errors that strike CPUs and registers during the execution of a job: the probability of having a silent error is determined solely by the number of flops of the job, or equivalently, by its sequential execution time. On the contrary, the amount of resources used to execute the job does not matter, even if the parallel execution time depends on the number of allocated processors. Suppose the occurrence of silent errors follows an exponential distribution with rate λ, then the failure probability for job J_j is given by:

$$q_j = 1 - e^{-\lambda t_j(1)} , \tag{2.3}$$

where $t_j(1)$ denotes the sequential execution time of job J_j. Then, the probability that the job fails f_j times before succeeding on the $f_j + 1$-st execution is $q_j(f_j) = q_j^{f_j}(1 - q_j)$. Assuming that errors occur independently for different jobs, the probability that a failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ happens can then be computed as $Q(\mathbf{f}) = \prod_{j=1}^{n} q_j(f_j)$.

[3] We consider the studied problem *offline*, although the failure scenario is unknown to the scheduler a prior and only revealed on-the-fly as jobs complete. One can also view the problem as *semi-online*, in which case all of our obtained approximation ratios can be interpreted as competitive ratios.

In general, given the probability $Q(\mathbf{f})$ of each failure scenario $\mathbf{f}$, we can define the *expected approximation ratio* of an algorithm ALG for a job set $\mathcal{J}$ as follows[4]:

$$\mathbb{E}\left[\frac{T_{\text{ALG}}(\mathcal{J})}{T_{\text{OPT}}(\mathcal{J})}\right] = \sum_{\mathbf{f}} Q(\mathbf{f}) \cdot \frac{T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} \ , \tag{2.4}$$

and the algorithm is said to be *r-approximation in expectation* if its expected approximation ratio is at most r for any job set $\mathcal{J}$, i.e.,

$$\sup_{\mathcal{J}} \ \mathbb{E}\left[\frac{T_{\text{ALG}}(\mathcal{J})}{T_{\text{OPT}}(\mathcal{J})}\right] = r \ . \tag{2.5}$$

While the approximation ratio of a scheduling algorithm under any failure scenario shows its worst-case performance, the expected approximation ratio shows its average-case performance. Clearly, a worst-case ratio will translate directly to the average case, because if the ratio holds true for every failure scenario, it is also true for the weighted sum. However, the converse may not be the case: an algorithm could have a very good expected approximation ratio, but perform arbitrarily worse than the optimal in some (low probability) failure scenarios.

In the theoretical analysis (Section 2.4), we mainly focus on bounding the worst-case approximation ratios of the proposed algorithms (except in Section 2.4.6, where we study the average-case performance of the BATCH-LIST algorithm). For the experimental evaluations (Section 2.6), we will instantiate the failure model with the silent error probability for each job as defined in Equation (2.3), and report both worst-case and average-case performance of the algorithms under a variety of experimental scenarios.

2.4 Resilient Scheduling Algorithms

In this section, we present two resilient scheduling algorithms (LPA-LIST and BATCH-LIST), and derive their approximation ratios for some common speedup models.

2.4.1 A Lower Bound on the Makespan

We first consider a simple lower bound on the makespan of any scheduling algorithm under a given failure scenario. This generalizes the well-known lower bound [116, 159] for the failure-free case.

Let $\mathbf{p}$ denote the processor allocation decision made by a scheduling algorithm ALG for job set $\mathcal{J}$ under failure scenario $\mathbf{f}$. Then, we define, respectively, the *maximum cumulative*

[4]While we use *expectation of ratios* to define the average-case performance of an algorithm, some studies in stochastic scheduling and online algorithms (e.g., [106, 121]) have used *ratio of expectations*, i.e.,

$$\frac{\mathbb{E}(T_{\text{ALG}})}{\mathbb{E}(T_{\text{OPT}})} = \frac{\sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{\sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} \ .$$

This approach, however, has not been favored by recent studies, since $\mathbb{E}(T_{\text{ALG}})$ could be dominated by "a few" instances with large objective functions, thus the ratio may not reflect the actual performance of the algorithm for "most" instances. See [137, 147] for a discussion on the two approaches.

execution time and *total cumulative area* of the jobs under algorithm ALG to be:

$$t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \max_{1 \le j \le n} \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \ , \tag{2.6}$$

$$A(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \sum_{j=1}^{n} \sum_{i=1}^{f_j+1} a_j(p_j^{(i)}) \ . \tag{2.7}$$

The following quantity serves as a lower bound on the makespan of the algorithm for job set $\mathcal{J}$ under failure scenario $\mathbf{f}$:

$$L(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \max\left(t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}), \frac{A(\mathcal{J}, \mathbf{f}, \mathbf{p})}{P}\right) . \tag{2.8}$$

Thus, we have:

$$T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) \ge L(\mathcal{J}, \mathbf{f}, \mathbf{p}) \ , \tag{2.9}$$

regardless of the scheduling decision $\mathbf{s}$ of the algorithm.

2.4.2 Lpa-List Scheduling Algorithm

Our first algorithm, called LPA-LIST, adopts a *two-phase* approach [116, 159]. The first phase uses a *Local Processor Allocation* (LPA) strategy to decide processor allocation $\mathbf{p}$ of the jobs, and the second phase uses LIST scheduling to determine the starting time $\mathbf{s}$ of the jobs.

List Scheduling Strategy

We first discuss LIST scheduling for the second phase, assuming a given processor allocation $\mathbf{p}$. Algorithm 1 shows the pseudocode.

The strategy first organizes all jobs in a list based on some priority. Then, at time 0 or whenever a running job J_k completes and hence releases processors, the algorithm detects if job J_k has errors. If so, the job will be inserted back into the list, again based on its priority, to be re-scheduled later. It finally scans the list of pending jobs and schedules all jobs that can be executed at the current time with the available processors. We point out that the algorithm essentially resembles a greedy backfilling strategy. In our analysis below, we will show that the worst-case approximation ratio is independent of the job priorities used, although it may affect the algorithm's practical performance. In Section 2.6, we will consider some commonly used priority rules for the experimental evaluation.

The following lemma shows the worst-case performance of the LIST scheduling strategy. Note that the job set $\mathcal{J}$ is dropped from the notations since the context is clear.

Lemma 1. *Given a processor allocation decision* $\mathbf{p}$ *for the jobs, the makespan of a* LIST *schedule (that determines the starting times* $\mathbf{s}$*) under any failure scenario* $\mathbf{f}$ *satisfies:*

$$T_{\text{LIST}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) \le \begin{cases} \frac{2A(\mathbf{f},\mathbf{p})}{P}, & \text{if } p_{\min} \ge \frac{P}{2} \\ \frac{A(\mathbf{f},\mathbf{p})}{P - p_{\min}} + \frac{(P-2p_{\min}) \cdot t_{\max}(\mathbf{f},\mathbf{p})}{P - p_{\min}}, & \text{if } p_{\min} < \frac{P}{2} \end{cases}$$

where $p_{\min} \ge 1$ *denotes the minimum number of utilized processors at any time during the schedule.*

Algorithm 1: LIST (Scheduling Strategy)

```
begin
    Organize all jobs in a list L according to some priority rule
    P_avail ← P
    f_j ← 0, ∀j
    when at time 0 or a running job J_k completes execution do
        P_avail ← P_avail + p_k^(f_k+1)
        if job J_k failed then
            L.insert_with_priority(J_k)
            f_k ← f_k + 1
        for j = 1, ..., |L| do
            J_j ← L(j)
            if P_avail ≥ p_j^(f_j+1) then
                execute job J_j at the current time
                P_avail ← P_avail − p_j^(f_j+1)
                L.remove(J_j)
```

Proof. We first observe that LIST only allocates and de-allocates processors upon job completions. Hence, the entire schedule can be divided into a set of consecutive and non-overlapping intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$, where jobs start (or complete) at the beginning (or end) of an interval, and v denotes the total number of intervals. Let $|I_\ell|$ denote the length of interval I_ℓ. The makespan under a failure scenario $\mathbf{f}$ can then be expressed as $T_{\text{LIST}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) = \sum_{\ell=1}^{v} |I_\ell|$.

Let $p(I_\ell)$ denote the number of utilized processors during an interval I_ℓ. Since the minimum number of utilized processors during the schedule is $p_{\min}$, we have $p(I_\ell) \geq p_{\min}$ for all $I_\ell \in \mathcal{I}$. We consider the following two cases:

Case 1: $p_{\min} \geq \frac{P}{2}$. In this case, we have $p(I_\ell) \geq p_{\min} \geq \frac{P}{2}$ for all $I_\ell \in \mathcal{I}$. Based on the definition of total cumulative area, we have $A(\mathbf{f}, \mathbf{p}) = \sum_{\ell=1}^{v} |I_\ell| \cdot p(I_\ell) \geq \frac{P}{2} \cdot T_{\text{LIST}}(\mathbf{f}, \mathbf{p}, \mathbf{s})$. This implies that:

$$T_{\text{LIST}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) \leq \frac{2A(\mathbf{f}, \mathbf{p})}{P} \ .$$

Case 2: $p_{\min} < \frac{P}{2}$. Let $I_{\min}$ denote the last interval in the schedule with processor utilization $p_{\min}$, and consider a job J_j that is running during interval $I_{\min}$. Necessarily, we have $p_j \leq p_{\min}$. We now divide the set $\mathcal{I}$ of intervals into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$, where $\mathcal{I}_1$ contains the intervals in which job J_j is running (including all of its execution attempts), and $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$. Let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$ and $T_2 = \sum_{I \in \mathcal{I}_2} |I|$ denote the total lengths of all intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Based on the definition of maximum cumulative execution time, we have $T_1 = \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \leq t_{\max}(\mathbf{f}, \mathbf{p})$.

For any interval $I \in \mathcal{I}_2$ that lies between the i-th execution attempt and the $(i+1)$-th execution attempt of J_j in the schedule, where $0 \leq i \leq f_j$, the processor utilization of I must satisfy $p(I) > P - p_{\min}$, since otherwise there are at least $p_{\min} \geq p_j$ available processors during interval I and hence the $i+1$-st execution attempt of J_j would have been scheduled at the beginning of I.

For any interval $I \in \mathcal{I}_2$ that lies after the (f_j+1)-th (last) execution attempt of J_j, there must be a job J_k running during I and that was not running during $I_{\min}$ (meaning no attempt of executing J_k was made during $I_{\min}$). This is because $p(I) > p_{\min}$, hence the job configuration must differ between I and $I_{\min}$. The processor utilization during interval I must also satisfy $p(I) > P - p_{\min}$, since otherwise the processor allocation of

J_k will be $p_k \leq p(I) \leq P - p_{\min}$, implying that the first execution attempt of J_k after interval $I_{\min}$ would have been scheduled at the beginning of $I_{\min}$.

Thus, for all $I \in \mathcal{I}_2$, we have $p(I) > P - p_{\min}$. Based on the definition of total cumulative area, we have $A(\mathbf{f}, \mathbf{p}) \geq (P - p_{\min}) \cdot T_2 + p_{\min} \cdot T_1$. The makespan of List under failure scenario $\mathbf{f}$ can then be derived as:

$$\begin{aligned} T_{\text{List}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) &= T_1 + T_2 \\ &\leq T_1 + \frac{A(\mathbf{f}, \mathbf{p}) - p_{\min} \cdot T_1}{P - p_{\min}} \\ &= \frac{A(\mathbf{f}, \mathbf{p})}{P - p_{\min}} + \frac{(P - 2p_{\min}) \cdot T_1}{P - p_{\min}} \\ &\leq \frac{A(\mathbf{f}, \mathbf{p})}{P - p_{\min}} + \frac{(P - 2p_{\min}) \cdot t_{\max}(\mathbf{f}, \mathbf{p})}{P - p_{\min}} \ . \end{aligned}$$

□

While Lemma 1 bounds the general performance of a List schedule for a given processor allocation $\mathbf{p}$, the following lemma shows its approximation ratio when the processor allocation strategy satisfies certain properties.

Lemma 2. *Given any failure scenario $\mathbf{f}$, if the processor allocation decision $\mathbf{p}$ satisfies:*

$$\begin{aligned} A(\mathbf{f}, \mathbf{p}) &\leq \alpha \cdot A(\mathbf{f}, \mathbf{p}^*) \ , \\ t_{\max}(\mathbf{f}, \mathbf{p}) &\leq \beta \cdot t_{\max}(\mathbf{f}, \mathbf{p}^*) \ , \end{aligned}$$

where $\mathbf{p}^*$ denotes the processor allocation of an optimal schedule, then a List schedule using processor allocation $\mathbf{p}$ is $r(\alpha, \beta)$-approximation, where

$$r(\alpha, \beta) = \begin{cases} 2\alpha, & \text{if } \alpha \geq \beta \\ \frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta, & \text{if } \alpha < \beta \end{cases}$$

Proof. Based on Lemma 1, when $p_{\min} \geq \frac{P}{2}$, we have:

$$T_{\text{List}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) \leq \frac{2A(\mathbf{f}, \mathbf{p})}{P} \leq \frac{2\alpha \cdot A(\mathbf{f}, \mathbf{p}^*)}{P} \leq 2\alpha \cdot T_{\text{Opt}}(\mathbf{f}, \mathbf{p}^*, \mathbf{s}^*).$$

The last inequality above is due to the makespan lower bound, as shown in Equation (2.8).

When $p_{\min} < \frac{P}{2}$, we can derive:

$$\begin{aligned} T_{\text{List}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) &\leq \frac{A(\mathbf{f}, \mathbf{p})}{P - p_{\min}} + \frac{(P - 2p_{\min}) \cdot t_{\max}(\mathbf{f}, \mathbf{p})}{P - p_{\min}} \\ &\leq \frac{\alpha \cdot A(\mathbf{f}, \mathbf{p}^*)}{P - p_{\min}} + \frac{\beta(P - 2p_{\min}) \cdot t_{\max}(\mathbf{f}, \mathbf{p}^*)}{P - p_{\min}} \\ &\leq \frac{(\alpha + \beta)P - 2\beta p_{\min}}{P - p_{\min}} \cdot T_{\text{Opt}}(\mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \\ &= \left(\alpha + \beta + (\alpha - \beta)\frac{p_{\min}}{P - p_{\min}}\right) \cdot T_{\text{Opt}}(\mathbf{f}, \mathbf{p}^*, \mathbf{s}^*). \end{aligned}$$

We have $\frac{1}{P-1} \leq \frac{p_{\min}}{P - p_{\min}} < 1$, since $1 \leq p_{\min} < \frac{P}{2}$. Therefore, if $\alpha \geq \beta$, we get:

$$T_{\text{List}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) \leq 2\alpha \cdot T_{\text{Opt}}(\mathbf{f}, \mathbf{p}^*, \mathbf{s}^*),$$

and if $\alpha < \beta$, we get:

$$T_{\text{List}}(\mathbf{f}, \mathbf{p}, \mathbf{s}) \leq \left(\frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta\right) \cdot T_{\text{Opt}}(\mathbf{f}, \mathbf{p}^*, \mathbf{s}^*).$$

Note that, in this case, $\frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta > 2\alpha$. □

Local Processor Allocation (Lpa)

The previous lemma is dealing with the global processor allocation, and in this section we will extend those results to local choices, because the number of repetitions of each job is unknown.

Lemma 3. *Given any failure scenario* $\mathbf{f}$*, and a collection* $\mathbf{p}$*, if there exists* (α,β) *such that for all job* J_j *and execution attempt* i*,* $\frac{t_j(p_j^{(i)})}{t_{j,\min}} \leq \beta$ *and* $\frac{a_j(p_j^i)}{a_{j,\min}} \leq \alpha$*, where* $t_{j,\min}$ *(resp.* $a_{j,\min}$*) is the minimum time of execution (resp. area) obtainable out of all the possible allocation, then a* List *schedule using processor allocation* $\mathbf{p}$ *is* $r(\alpha,\beta)$*-approximation, where*

$$r(\alpha,\beta) = \begin{cases} 2\alpha, & \text{if } \alpha \geq \beta \\ \frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta, & \text{if } \alpha < \beta \end{cases}, \tag{2.10}$$

where $\mathbf{p}^*$ *denotes the processor allocation of an optimal schedule.*

Proof. We consider the job J_j that achieves the *maximum cumulative execution time* i.e. such that

$$t_{\max}(\mathbf{f},\mathbf{p}) = \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \ ,$$

Then

$$\begin{aligned} t_{\max}(\mathbf{f},\mathbf{p}) = \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \leq \sum_{i=1}^{f_j+1} \beta t_{j,\min} \leq \beta \sum_{i=1}^{f_j+1} t_j(p_j^{*(i)}) \\ \leq \beta \max_{1\leq k\leq n} \sum_{i=1}^{f_k+1} t_k(p_k^{*(i)}) = \beta t_{\max}(\mathbf{f},\mathbf{p}^*), \end{aligned}$$

Similarly,

$$A(\mathbf{f},\mathbf{p}) = \sum_{j=1}^{n}\sum_{i=1}^{f_j+1} a_j(p_j^{(i)}) \leq \sum_{j=1}^{n}\sum_{i=1}^{f_j+1} \alpha a_{j,\min} \leq \alpha \sum_{j=1}^{n}\sum_{i=1}^{f_j+1} a_j(p_j^{*(i)}) \leq A(\mathbf{f},\mathbf{p}^*) \ .$$

This shows that the condition are verified, therefore we can apply Lemma 2 to conclude. □

We now discuss the Lpa strategy for the first phase of the algorithm. Given the result of Lemma 3, Lpa uses a parameter α^{max} allocates processors locally for each job and its pseudocode is shown in Algorithm 2. For each job J_j, the strategy first computes its minimum possible execution time and area. Then, it chooses a processor allocation that leads to the smallest ratio β_j while respecting the constraints on α^{max}. The choice of α^{max} will depend on the model and will be discussed in the next section. More specifically, we will give for each speedup models an α^{max} for which there always exists an allocation such that $\beta_j \leq \alpha^{max}$, and therefore using Lemma 3, we will show that List (α^{max}) is a $2\alpha^{max}$ approximation for the given speedup model. The α^{max} are given in Figure 2.1

Once the processor allocation of a job has been decided, the same allocation will be used by the List scheduling strategy in the second phase throughout the execution until the job completes successfully without failures.

Model	Roofline	Communication	Amdahl	Mix	Power	Monotonic
α^{max}	1	1.5	2	27/13	$P^{1/4}$	$\sqrt{P}$
Ratio	2	3	4	$54/13 \approx 4.15$	$O(P^{1/4})$	$O(\sqrt{P})$

Figure 2.1: Parameters for each speedup model

Algorithm 2: LPA (α^{max}) (Processor Allocation Strategy)

```
begin
    for j = 1, 2, ..., n do
        t_min ← ∞, a_min ← ∞
        for p = 1, 2, ..., P do
            if t_j(p) < t_min then
                t_min ← t_j(p)
            if p · t_j(p) < a_min then
                a_min ← p · t_j(p)
        p_j ← 0, β_min ← ∞
        for p = 1, 2, ..., P do
            α_j ← p · t_j(p)/a_min
            β_j ← t_j(p)/t_min
            if α_j < α^max and β_j < β_min then
                p_j ← p, β_min ← β_j
```

2.4.3 Worst-Case Performance of Lpa-List for Some Common Speedup Models

We now analyze the worst-case performance of the LPA-LIST algorithm for moldable jobs that exhibit some common speedup models, as well as for the general monotonic model. All derived approximation ratios are independent of the failure scenarios, hence based on Equations (2.4) and (2.5). The same ratios also apply to the average-case performance of the algorithm for the respective speedup models.

In the following, we first consider the three special speedup models (i.e., roofline, communication and Amdahl) before tackling the mix model, and we finish with power and monotonic models. For clarity, we introduce superscript $M \in \{\text{ROO}, \text{COM}, \text{AMD}, \text{MIX}\}$ to the notations α^M and β^M corresponding to Equation (2.10). Given a speedup model M, the analysis focuses on finding α^M and β^M for each individual task, thus we will drop the task index j for simplicity.

Roofline Model

In the roofline model, the execution time of a job J when allocated p processors satisfies $t(p) = \frac{w}{\min(p,\bar{p})}$ for a bounded degree of parallelism $1 \leq \bar{p} \leq P$.

Theorem 1. *LPA-LIST (1) is a 2-approximation for jobs with the roofline speedup model.*

Proof. In the roofline speedup model, the minimum execution time of a job J is $t_{\min} = w/\bar{p}$ and the minimum area of the job is $a_{\min} = w$. These two quantities can be achieved by simply allocating $p = \bar{p}$ processors to the job. This leads to the bounds of $\alpha = 1$ and $\beta = 1$ for each job as well as globally under any failure scenario. Hence, based on Lemma 3, we get an approximation ratio of $2\alpha = 2$. □

Communication Model

In the communication model [54, 78], the execution time of a job J when allocated p processors is given by $t(p) = w/p + (p-1)c$, where $c \geq 0$ denotes the per-processor communication overhead.

Lemma 4. *For any task that follows the communication model, there exists a processor allocation that achieves* $\alpha^{\text{COM}} = \frac{4}{3}$ *and* $\beta^{\text{COM}} = \frac{3}{2}$.

Proof. For simplicity, we rewrite $t(p) = w/p+(p-1)c = c\,(w'/p + p - 1)$. We note $p^{\max}$ the number of processors that would minimize the task's execution time $t(p)$ if P was infinite, i.e., $t(p^{\max}) = t_*^{\min} \leq t^{\min}$. Clearly, we have $\lfloor\sqrt{w'}\rfloor \leq p^{\max} \leq \lceil\sqrt{w'}\rceil$. By derivation, $\sqrt{w'}$ minimizes the time function, which is decreasing then increasing, therefore the integer minimizing the function is the lower or higher integer part of $\sqrt{w'}$. Also, the minimum area of the task is obtained with one processor, i.e., $a^{\min} = a(1) = cw'$.

Furthermore, for a given choice of p, we can show that $f(w', p) \triangleq \frac{t(p)}{t_*^{\min}} = \frac{\frac{w'}{p}+p-1}{\frac{w'}{p^{\max}}+p^{\max}-1}$ is a non-decreasing function of w' in the interval $[p^2, \infty)$. To see that, we can check it is continuous and compute the partial derivative:

$$\begin{aligned}\frac{\partial f(w',p)}{\partial w'} &= \frac{\frac{1}{p}\left(\frac{w'}{p^{\max}} + p^{\max} - 1\right) - \frac{1}{p^{\max}}\left(\frac{w'}{p} + p - 1\right)}{\left(\frac{w'}{p^{\max}} + p^{\max} - 1\right)^2} \\ &= \frac{\frac{p^{\max}-1}{p} - \frac{p-1}{p^{\max}}}{\left(\frac{w'}{p^{\max}} + p^{\max} - 1\right)^2},\end{aligned}$$

which is defined everywhere except at the points where $p^{\max}$ changes (due to changes of w'), and has the same sign as $\frac{p^{\max}-1}{p} - \frac{p-1}{p^{\max}} \geq 0$. The last inequality is because if $p = p^{\max}$, it is equal to 0, otherwise $\frac{p^{\max}-1}{p} \geq 1$ and $\frac{p-1}{p^{\max}} < 1$. This remains true if $p \leq p^{\max}$, which is satisfied when $w' \geq p^2$.

We now consider three cases:

Case 1: $w' \leq 6$. In this case, we set $p = 1$, which gives the minimum area, i.e., $\frac{a(p)}{a^{\min}} = 1$. When $w' \leq 1$, setting $p = 1$ also gives the minimum execution time, i.e., $\frac{t(p)}{t^{\min}} = 1$. Otherwise, we have $\frac{t(p)}{t^{\min}} \leq f(w', 1) \leq f(6, 1) = \frac{6}{\min(\frac{6}{2}+1, \frac{6}{3}+2)} = \frac{3}{2}$, since $p^{\max} = 2$ or $p^{\max} = 3$ when $w' = 6$.

Case 2: $6 < w' \leq 25$. In this case, we set $p = 2$ and get $\frac{a(p)}{a^{\min}} = \frac{c(w'+2)}{cw'} = 1 + \frac{2}{w'} < \frac{4}{3}$. As $w' > p^2 = 4$, we can also get $\frac{t(p)}{t^{\min}} = f(w', 2) \leq f(25, 2) = \frac{\frac{25}{2}+1}{\frac{25}{5}+4} = \frac{27}{18} = \frac{3}{2}$, since $p^{\max} = 5$ when $w' = 25$.

Case 3: $w' > 25$. In this case, we have $t^{\min} \geq c(2\sqrt{w'} - 1)$, which is the minimum possible execution time if the processor allocation could be non-integers. We set $p = \left\lfloor\sqrt{\frac{w'}{3}} + \frac{1}{2}\right\rfloor$ and obtain $\frac{a(p)}{a^{\min}} = \frac{c(w'+p(p-1))}{cw'} \leq 1 + \frac{1}{w'}\left(\sqrt{\frac{w'}{3}} + \frac{1}{2}\right)\left(\sqrt{\frac{w'}{3}} - \frac{1}{2}\right) \leq 1 + \frac{1}{w'}\frac{w'}{3} = \frac{4}{3}$. Finally, $\frac{t(p)}{t^{\min}} \leq \frac{c\left(\frac{w'}{\sqrt{\frac{w'}{3}}-\frac{1}{2}} + \sqrt{\frac{w'}{3}}\right)}{c(2\sqrt{w'}-1)} = \frac{1}{2-\frac{1}{\sqrt{w'}}}\left(\frac{1}{\frac{1}{\sqrt{3}}-\frac{1}{2\sqrt{w'}}} + \frac{1}{\sqrt{3}}\right)$. This function is clearly decreasing with w', and using $w' > 25$, we get $\frac{t(p)}{t^{\min}} \leq \frac{5}{9}\left(\frac{10\sqrt{3}}{10-\sqrt{3}} + \frac{1}{\sqrt{3}}\right) \approx 1.48 < \frac{3}{2}$. □

Theorem 2. *LPA-LIST (1.5) is a 3-approximation for jobs with the communication model.*

Proof. This results directly comes from Lemma 4 and Lemma 2, because if we have shown the existence of the bound for $\alpha^{\text{Com}} = \frac{4}{3}$, it holds in particular for α^{Com}. □

Remarks. Our result improves upon the 4-approximation of the Set algorithm [78], which is the best ratio known for this model. Our result further extends the one in [78] in two ways: (1) The model in [78] assumes the same communication overhead c for all jobs, while we consider an individual overhead c for each job J; (2) The algorithm in [78] applies to failure-free job executions, while our algorithm is able to handle job failures.

Amdahl's Model

In Amdahl's model [3], the execution time of a job J when allocated p processors satisfies $t(p) = w(\frac{1-d}{p} + d)$, where $d \in [0, 1]$ denotes the inherently sequential fraction of the job. It is a particular case of the monotonic model as described in Section 2.3.1. For convenience, we consider an equivalent form of the model in the analysis: $t(p) = \frac{w}{p} + d$, where w denotes the parallelizable work of the job and d denotes the inherently sequential work.

Lemma 5. *For any $\alpha > 1$, there exists a processor allocation p that satisfies the α bound, i.e., $\frac{a(p)}{a^{\min}} \leq \alpha$ and at the same time achieves $\beta(\alpha) = \frac{\alpha}{\alpha-1}$, i.e., $\frac{t(p)}{t^{\min}} \leq \beta(\alpha) = \frac{\alpha}{\alpha-1}$*

Proof. The minimum execution time of the task is obtained with all P processors, i.e., $t^{\min} = t(P) = \frac{w}{P} + d$, and the minimum area with just one processor, i.e., $a^{\min} = a(1) = w + d$.

To show the result, for any given $\alpha > 1$, we let $x = \alpha - 1$, and set $p = \min(\lceil x\frac{w}{d} \rceil, P)$. This implies $p \leq \lceil x\frac{w}{d} \rceil \leq x\frac{w}{d} + 1$. Thus, we have $\frac{a(p)}{a^{\min}} = \frac{w+dp}{w+d} \leq \frac{w+d\left(x\frac{w}{d}+1\right)}{w+d} = \frac{w+d+xw}{w+d} = 1 + \frac{xw}{w+d} \leq 1 + x = \alpha$. Furthermore, if $p = \lceil x\frac{w}{d} \rceil \geq x\frac{w}{d}$, we have $\frac{t(p)}{t^{\min}} \leq \frac{\frac{w}{x\frac{w}{d}}+d}{\frac{w}{P}+d} \leq \frac{\frac{d}{x}+d}{d} = \frac{1}{x} + 1 = \frac{1}{\alpha-1} + 1 = \frac{\alpha}{\alpha-1} = \beta(\alpha)$. Otherwise, if $p = P$, we get $t(p) = t^{\min}$ and thus $\frac{t(p)}{t^{\min}} = 1 < \frac{\alpha}{\alpha-1} = \beta(\alpha)$. □

Theorem 3. *Lpa-List (2) is a 4-approximation for jobs with the Amdahl's speedup model.*

Proof. We use Lemma 2 altogether with Lemma 5 using $\alpha = 2$, to get an approximation ratio of $2\alpha = 4$. □

Mix Model

We now consider the mixed model combining Roofline, Communication and Amdahl's models as follows: $t(p) = \frac{w(1-d)}{\min(p,\bar{p})} + wd + (p-1)c$, which could capture more realistically the speedups of some complex applications. In this model, we only need to consider $p \leq \bar{p}$, since any $p > \bar{p}$ will obviously be a bad choice. To simplify the analysis, we also factorize the function by c and obtain the following equivalent form: $t(p) = c\Big(\frac{w'}{p} + d' + (p-1)\Big)$, with $w' = \frac{w(1-d)}{c}$ and $d' = \frac{wd}{c}$.

Lemma 6. *For any task that follows the mix model, there exists a processor allocation that achieves $\alpha^{\text{Mix}} = 2$ and $\beta^{\text{Mix}} = \frac{27}{13}$.*

Proof. If we allow the processor allocation to take non-integer values and assuming unbounded $\bar{p}$, the execution time function $t(p)$ would be minimized at $p^* = \sqrt{w'}$. Thus, the minimum execution time should satisfy $t^{\min} \geq c(2\sqrt{w'} + d' - 1)$. Note that this bound will hold true regardless of the value of $\bar{p}$: it is obviously true if $\bar{p} \geq p^*$, otherwise $t^{\min}$ is

achieved at $\bar{p}$, with a value also higher than $c(2\sqrt{w'}+d'-1)$. Furthermore, the minimum area is obtained with one processor, i.e., $a^{\min}=a(1)=c(w'+d')$.

Again, we let $p^{\max}$ denotes the number of processors that minimizes the execution time if we had an infinite number of processors, i.e., $t(p^{\max})\leq t^{\min}$. Clearly, we have either $p^{\max}=\bar{p}$ or $\lfloor\sqrt{w'}\rfloor\leq p^{\max}\leq\lceil\sqrt{w'}\rceil$.

We consider three cases.

Case 1: $w'\leq 4$ or $\bar{p}=1$. In this case, we must have $p^{\max}\leq 2$. We can then set $p=1$, and get $\frac{a(p)}{a^{\min}}=1$ and $\frac{t(p)}{t^{\min}}\leq 2$.

Case 2: $4<w'\leq 49$ and $\bar{p}\geq 2$. In this case, we set $p=2$ and get $\frac{a(p)}{a^{\min}}\leq\frac{w'+2d'+2}{w'+d'}\leq 2$. Similarly to the proof of Lemma 4 (for the communication model), we can show that $f(w',p)\triangleq\frac{t(p)}{t^{\min}}=\frac{\frac{w'}{p}+d'+p-1}{\frac{w'}{p^{\max}}+d'+p^{\max}-1}$ is increasing with w' if $w'\geq p^2$. Therefore, we can get $\frac{t(p)}{t^{\min}}\leq f(49,2)\leq\frac{\frac{49}{2}+d'+1}{2\sqrt{49}+d'-1}=\frac{51+2d'}{26+2d'}\leq 2$.

Case 3: $w'>49$ and $\bar{p}\geq 2$. In this case, we will set $p=\min\left(\left\lfloor\frac{w'+d'}{\sqrt{w'}+d'}+\frac{1}{2}\right\rfloor,\bar{p}\right)$ and get:

$$\begin{aligned}
\frac{a(p)}{a^{\min}} &= \frac{w'+p(d'+p-1)}{w'+d'}\\
&\leq \frac{w'+\left(\frac{w'+d'}{\sqrt{w'}+d'}+\frac{1}{2}\right)\left(d'+\frac{w'+d'}{\sqrt{w'}+d'}-\frac{1}{2}\right)}{w'+d'}\\
&= \frac{w'+\frac{d'}{2}-\frac{1}{4}+\frac{w'+d'}{\sqrt{w'}+d'}\left(d'+\frac{w'+d'}{\sqrt{w'}+d'}\right)}{w'+d'}\\
&\leq \frac{w'+d'+\frac{w'+d'}{\sqrt{w'}+d'}\left(d'+\frac{w'+d'}{\sqrt{w'}+d'}\right)}{w'+d'}\\
&= 1+\frac{d'(\sqrt{w'}+d')+w'+d'}{(\sqrt{w'}+d')^2}\\
&= 1+\frac{d'^2+d'\sqrt{w'}+d'+w'}{d'^2+2d'\sqrt{w'}+w'}\\
&\leq 2\ .
\end{aligned}$$

The last inequality above comes from $w'>1$ and $d'>0$.

Since $w'>1$, we get $t^{\min}\geq c(2\sqrt{w'}+d'-1)>c(\sqrt{w'}+d')$. To derive the execution time ratio, we further consider two subcases.

- If $p=\left\lfloor\frac{w'+d'}{\sqrt{w'}+d'}+\frac{1}{2}\right\rfloor$, then $p\geq\frac{w'+d'}{\sqrt{w'}+d'}-\frac{1}{2}\geq\frac{w'-\frac{1}{2}\sqrt{w'}}{\sqrt{w'}+d'}$. We can then get:

$$\begin{aligned}
\frac{t(p)}{t^{\min}} &\leq \frac{\frac{w'}{p}+d'+p-1}{\sqrt{w'}+d'}\\
&\leq \frac{\frac{w'(\sqrt{w'}+d')}{w'-\frac{1}{2}\sqrt{w'}}}{\sqrt{w'}+d'}+\frac{d'+\frac{w'+d'}{\sqrt{w'}+d'}}{\sqrt{w'}+d'}\\
&\leq \frac{1}{1-\frac{1}{2\sqrt{w'}}}+\frac{d'(\sqrt{w'}+d')+w'+d'}{(\sqrt{w'}+d')^2}\\
&\leq \frac{1}{1-\frac{1}{2\sqrt{w'}}}+1
\end{aligned}$$

For the last inequality, we recognize the same term we had when bounding the area ratio, which is at most 1. Finally, the last expression above decreases with w', so using $w' > 49$, we get $\frac{t(p)}{t^{\min}} \leq \frac{1}{1-\frac{1}{14}} + 1 = \frac{27}{13}$.

- If $p = \bar{p} < \left\lfloor \frac{w'+d'}{\sqrt{w'}+d'} + \frac{1}{2} \right\rfloor$, and since $\bar{p}$ is an integer, then it is necessarily the case that $\bar{p} \leq \left\lfloor \frac{w'+d'}{\sqrt{w'}+d'} + \frac{1}{2} \right\rfloor - 1 \leq \frac{w'+d'}{\sqrt{w'}+d'} \leq \sqrt{w'}$ (because $w' > 1$). Therefore, we should also have $p^{\max} = \bar{p} = p$, and thus $\frac{t(p)}{t^{\min}} = 1$. □

Theorem 4. *LPA-LIST (3) is a $\frac{54}{13}$-approximation for jobs with the mixed model.*

Proof. The result of Lemma 6 also holds with $\alpha = \beta = \frac{27}{13}$. Thus, with Lemma 2, we get an approximation ratio of $2\alpha = \frac{54}{13}$. □

Power Model

In the power model, the execution time of a job J_j when allocated p processors satisfies $t_j(p) = w_j/p^{\delta_j}$, where $\delta_j \in [0, 1]$ is a constant parameter. This speedup has been observed in some linear algebra applications [73, 148] and it is also an example of the monotonic model.

Theorem 5. *LPA-LIST ($P^{1/4}$ is a $\Theta(P^{1/4})$-approximation for jobs with the power model.*

Proof. In the power model, the minimum execution time of a job J_j is $t_{\min} = \frac{w_j}{P^{\delta_j}}$ (achieved by allocating P processors), and the minimum area of the job is $a_{\min} = w_j$ (achieved by allocating one processor).

By allocating p_j processors to the job, we will get $\alpha = \frac{a_j(p_j)}{a_{\min}} = p_j^{1-\delta_j}$ and $\beta = \frac{t_j(p_j)}{t_{\min}} = (\frac{P}{p_j})^{\delta_j}$. Hence, to minimize Equation (2.10), the algorithm will choose $p_j = \Theta(P^{\delta_j})$ processors, resulting in an approximation ratio of $\Theta(P^{\delta_j(1-\delta_j)})$. Since $\delta_j \in [0, 1]$, the value of $\delta_j(1 - \delta_j)$ is maximized at $\delta_j = 1/2$, leading to an approximation ratio of $\Theta(P^{1/4})$. □

Monotonic Model

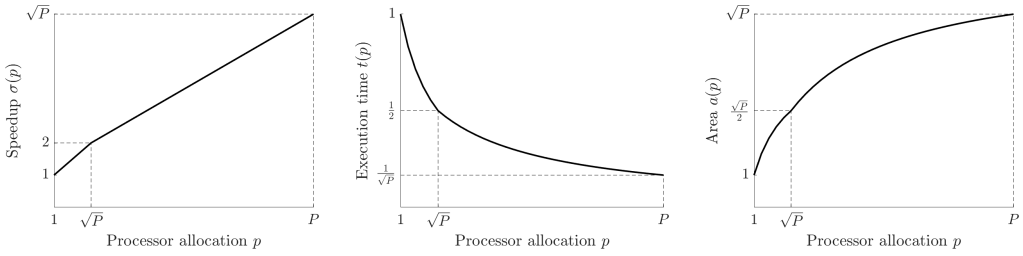
We now consider the general monotonic model. Recall that a job J_j is *monotonic*, if $t_j(p) \geq t_j(p')$ and $a_j(p) \leq a_j(p')$ for any $p \leq p'$. This means that the execution time of the job will not increase with the processor allocation and the area will not decrease with the processor allocation. In particular, the area assumption implies that the speedup efficiency of the job will not increase as more processors are allocated to it, i.e., $\sigma_j(p)/p \geq \sigma_j(p')/p'$, a property that has been observed in many practical parallel applications.

Theorem 6. *LPA-LIST ($\sqrt{P}$) is an $O(\sqrt{P})$-approximation for jobs with the monotonic model.*

Proof. In a general monotonic model, the minimum execution time of a job J_j is achieved with P processors, i.e., $t_{\min} = t_j(P)$, and the minimum area is achieved with one processor, i.e., $a_{\min} = a_j(1) = t_j(1)$.

Consider an allocation $p_j = \lfloor \sqrt{P} \rfloor$. Based on the monotonic assumption, we get $a_j(p_j) = p_j t_j(p_j) \leq \sqrt{P} \cdot t_j(1) = \sqrt{P} \cdot a_{\min}$, and $t_j(p_j) \leq \frac{P}{p_j} t_j(P) = O(\sqrt{P}) \cdot t_{\min}$. Thus, based on Lemma 2, we get an approximation ratio of $O(\sqrt{P})$. □

We show that the above ratio is asymptotically tight for any algorithm that makes *local* processor allocation decisions based on individual job characteristics. Examples of such



(a) (b) (c)

Figure 2.2: Speedup (a), execution time (b), and area (c) profiles of the job used in the proof of Theorem 7.

algorithms include the LPA algorithm considered in this chapter and the SET algorithm studied in [78]. The result holds even under the additional assumption that the speedup profiles of the jobs are *concave* [93] and that jobs do not fail. In the next section, we will propose another algorithm that overcomes this limitation by making *coordinated* processor allocation decisions for a set of jobs.

Theorem 7. *Any scheduling algorithm that relies on local processor allocation for each individual job is $\Omega(\sqrt{P})$-approximation with the monotonic model.*

Proof. Assume that $\sqrt{P}$ is an integer and $P \geq 4$. We consider a job with a concave speedup profile[5] that contains two piece-wise linear segments defined by three points: $\sigma(1) = 1$, $\sigma(\sqrt{P}) = 2$ and $\sigma(P) = \sqrt{P}$ (see Figure 2.2(a)). Suppose the execution time of the job with one processor is $t(1) = 1$. We can then derive the execution time profile of the job as follows (see Figure 2.2(b)):

$$t(p) = \begin{cases} \frac{\sqrt{P}-1}{p+\sqrt{P}-2} & \text{if } p \leq \sqrt{P}, \\ \frac{P-\sqrt{P}}{p(\sqrt{P}-2)+P} & \text{if } p > \sqrt{P}; \end{cases}$$

and the area profile of the job as follows (see Figure 2.2(c)):

$$a(p) = \begin{cases} \frac{p(\sqrt{P}-1)}{p+\sqrt{P}-2} & \text{if } p \leq \sqrt{P}, \\ \frac{p(P-\sqrt{P})}{p(\sqrt{P}-2)+P} & \text{if } p > \sqrt{P}. \end{cases}$$

The job is obviously monotonic.

Suppose there are n identical such jobs in the system, where n depends on the processor allocation algorithm (denoted as ALG). Since the jobs are identical and processors are allocated locally, the processor allocation p for each job should be the same. We consider two cases.

Case 1: If $p \leq \sqrt{P}$, then there is only $n = 1$ job. In this case, the algorithm has a makespan of $T_{\text{ALG}} \geq t(\sqrt{P}) = \frac{1}{2}$ and the optimal makespan is $T_{\text{OPT}} = t(P) = \frac{1}{\sqrt{P}}$ by allocating P processors to the job.

Case 2: If $p > \sqrt{P}$, then there are $n = P$ jobs. In this case, the makespan of the algorithm satisfies $T_{\text{ALG}} \geq \frac{n \cdot a(p)}{P} \geq a(\sqrt{P}) = \frac{\sqrt{P}}{2}$, and the optimal makespan is $T_{\text{OPT}} = 1$ by allocating one processor to each job.

Thus, in both cases, we have $\frac{T_{\text{ALG}}}{T_{\text{OPT}}} \geq \frac{\sqrt{P}}{2}$. □

[5]The speedup profile is concave because $\sigma'(p) = \frac{1}{\sqrt{P}-1}$ for any $p \in [1, \sqrt{P})$, and $\sigma'(p) = \frac{\sqrt{P}-2}{P-\sqrt{P}} < \frac{\sqrt{P}}{P-\sqrt{P}} = \frac{1}{\sqrt{P}-1}$ for any $p \in (\sqrt{P}, P]$.

2.4.4 Batch-List Scheduling Algorithm

We now present the second algorithm, called Batch-List. Unlike the Lpa-List algorithm, which allocates processors locally for each job, Batch-List coordinates the processor allocation decisions for different jobs. While not knowing the failure scenario in advance, the algorithm organizes the execution attempts of the jobs in multiple *batches*, where each batch executes the pending jobs (i.e., the jobs that have not been successfully completed so far) up to a certain number of attempts that doubles after each batch. The idea is inspired by the *doubling strategy* [44] that has been commonly applied in many online problems. The following describes the details of the Batch-List algorithm.

Let B_k denote the k-th batch created by the algorithm, where $k \geq 1$. Let n_k denote the number of pending jobs immediately before B_k starts, and let $\mathcal{J}_k = \{J_{k,1}, J_{k,2}, \ldots, J_{k,n_k}\}$ denote this set of pending jobs. For convenience, we define $g_k = 2^{k-1}$. In batch B_k, we allow each pending job $J_{k,j}$ to have at most $f_{k,j} = g_k - 1$ failures, i.e., each job is allowed to make g_k execution attempts in the batch; if the job is still not successfully completed after that, it will be handled by the next batch B_{k+1}. Let $\mathbf{f}_k = (f_{k,1}, f_{k,2}, \ldots, f_{k,n_k})$ denote this worst-case failure scenario for the jobs in batch B_k. Given $\mathbf{f}_k$, each job $J_{k,j}$ can be represented by a chain $J_{k,j}^{(1)} \rightarrow J_{k,j}^{(2)} \rightarrow \cdots \rightarrow J_{k,j}^{(g_k)}$ of g_k sub-jobs with linear precedence constraint, where each sub-job represents an execution attempt of $J_{k,j}$ in the batch. Thus, all sub-jobs in batch B_k form a set of n_k linear chains, one for each pending job.

To allocate processors for all the sub-jobs (or the different execution attempts of the pending jobs) in batch B_k, we adopt the pseudo-polynomial time algorithm, called Mt-Allotment, proposed in [110] for series-parallel precedence graphs (of which a set of independent linear chains is a special case). Specifically, the algorithm determines an allocation $p_{k,j}^{(m)}$ for each sub-job $J_{k,j}^{(m)}$ (or the m-th execution attempt of job $J_{k,j}$). Let $\vec{p}_{k,j} = (p_{k,j}^{(1)}, p_{k,j}^{(2)}, \ldots, p_{k,j}^{(f_{k,j}+1)})$ be the vector of processor allocations for job $J_{k,j}$, and let $\mathbf{p}_k = (\vec{p}_{k,1}, \vec{p}_{k,2}, \ldots, \vec{p}_{k,n_k})$ be the processor allocations for all jobs in batch B_k. The following lemma shows the property of the allocation $\mathbf{p}_k$ returned by Mt-Allotment for jobs with any arbitrary speedup model.

Lemma 7. *For any $\epsilon > 0$, Mt-Allotment can compute, with complexity polynomial in $1/\epsilon$, a processor allocation $\mathbf{p}_k$ for all jobs in batch B_k that approximates the minimum makespan lower bound as defined in Equation (2.8) as follows:*

$$L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \leq (1+\epsilon) \cdot \min_{\mathbf{p}} L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) \ . \tag{2.11}$$

We refer to [110] for a detailed description of the Mt-Allotment algorithm and its analysis[6]. Once the processor allocation $\mathbf{p}_k$ has been decided, Batch-List schedules all pending jobs in a batch B_k using the List strategy as shown in Algorithm 1, while restricting each job to execute at most g_k times. After batch B_k completes and if there are still pending jobs, the algorithm will create a new batch B_{k+1} to schedule the remaining pending jobs.

2.4.5 Worst-Case Performance of Batch-List for Arbitrary Speedup Model

We analyze the worst-case performance of Batch-List for moldable jobs with any arbitrary speedup model.

[6] In a nutshell, the algorithm uses dynamic programming to decide whether there exists an allocation $\mathbf{p}$ such that $L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) \leq (1+\epsilon) \cdot X$ for a positive integer bound X, and performs a binary search on X.

First, we define the following concept: a job set $\mathcal{J}'$ with failure scenario $\mathbf{f}'$ is said to be *dominated* by a job set $\mathcal{J}$ with failure scenario $\mathbf{f}$, denoted by $(\mathcal{J}', \mathbf{f}') \subseteq (\mathcal{J}, \mathbf{f})$, if for every job $J_j \in \mathcal{J}'$, we have $J_j \in \mathcal{J}$ and $f'_j \leq f_j$. The following lemma gives two trivial properties without proof for a dominated pair of job set and failure scenario.

Lemma 8. *If $(\mathcal{J}', \mathbf{f}') \subseteq (\mathcal{J}, \mathbf{f})$, then we have:*
(a) $L(\mathcal{J}', \mathbf{f}', \mathbf{p}) \leq L(\mathcal{J}, \mathbf{f}, \mathbf{p})$;
(b) $T_{OPT}(\mathcal{J}', \mathbf{f}', \mathbf{p}'^*, \mathbf{s}'^*) \leq T_{OPT}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)$.

Lemma 9. *Suppose a job set $\mathcal{J}$ with failure scenario $\mathbf{f}$ is executed by* Batch-List*. Then, any job $J_j \in \mathcal{J}$ will successfully complete in $b_j = \lceil \log_2(f_j + 2) \rceil$ batches, and in any batch B_k, where $1 \leq k \leq b_j$, we have $f_{k,j} \leq f_j$.*

Proof. Since the algorithm allows the number of execution attempts of a job to double in each new batch, the maximum number of execution attempts of the job in a total of b batches is given by $\sum_{k=1}^{b} 2^{k-1} = 2^b - 1$. Thus, if a job J_j fails f_j times (i.e., executes $f_j + 1$ times), then the number of batches it takes to complete the job is $b_j = \lceil \log_2(f_j + 2) \rceil = 1 + \lfloor \log_2(f_j + 1) \rfloor$.

In any batch B_k until job J_j completes, where $1 \leq k \leq b_j$, we have $f_{k,j} = 2^{k-1} - 1 \leq 2^{\lfloor \log_2(f_j+1) \rfloor} - 1 \leq f_j$. □

The following theorem shows the approximation ratio of Batch-List for jobs with arbitrary speedup model.

Theorem 8. *Batch-List is an $O((1+\epsilon)\log_2(f_{\max}))$-approximation for jobs with arbitrary speedup model, where $f_{\max} = \max_j f_j$ denotes the maximum number of failures of any job in a failure scenario.*

Proof. According to Lemma 9, the total number of batches for any job set $\mathcal{J}$ with failure scenario $\mathbf{f}$ is given by $b_{\max} = \lceil \log_2(f_{\max} + 2) \rceil$. Further, for any batch B_k, where $1 \leq k \leq b_{\max}$, we have $(\mathcal{J}_k, \mathbf{f}_k) \subseteq (\mathcal{J}, \mathbf{f})$.

Let $\mathbf{f}'_k = (f'_{k,1}, f'_{k,2}, \ldots, f'_{k,n_k})$ denote the actual failure scenario for the jobs in batch B_k. Clearly, we have $f'_{k,j} \leq f_{k,j}$ for any $J_j \in \mathcal{J}_k$, and thus, $(\mathcal{J}_k, \mathbf{f}'_k) \subseteq (\mathcal{J}_k, \mathbf{f}_k)$.

Since Batch-List uses the Mt-Allotment algorithm to allocate processors and the List strategy to schedule all jobs in each batch, according to Lemmas 1, 7 and 8, we can bound the execution time of any batch B_k as follows:

$$\begin{aligned} T_{\text{List}}(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k, \mathbf{s}_k) &\leq 2 \cdot L(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k) \\ &\leq 2 \cdot L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \\ &\leq 2(1+\epsilon) \cdot L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}^*_k) \\ &\leq 2(1+\epsilon) \cdot T_{\text{OPT}}(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}^*_k, \mathbf{s}^*_k) \\ &\leq 2(1+\epsilon) \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \ . \end{aligned}$$

Therefore, the makespan of Batch-List satisfies:

$$\begin{aligned} T_{\text{Batch-List}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) &= \sum_{k=1}^{b_{\max}} T_{\text{List}}(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k, \mathbf{s}_k) \\ &\leq 2(1+\epsilon)\lceil \log_2(f_{\max} + 2) \rceil \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \ . \end{aligned}$$

□

We now show that the approximation ratio of Batch-List is tight up to a constant factor.

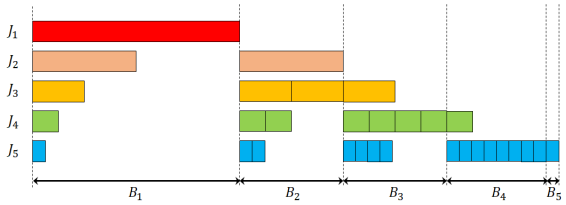


Figure 2.3: An illustration of the lower bound instance for the Batch-List algorithm shown in Theorem 9 with $K = 5$ jobs.

Theorem 9. *Batch-List is $\Omega(\log_2(f_{\max}))$-approximation.*

Proof. We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_K\}$ of K jobs and at least as many processors, so that each job can be executed on a dedicated processor. For each job J_j, where $1 \leq j \leq K$, its (sequential) execution time is $t_j = \frac{1}{2^j}$, and it fails $f_j = 2^{j-1} - 1$ times (i.e., executes 2^{j-1} times). Given this failure scenario $\mathbf{f}$, the total time to complete job J_j is given by $2^{j-1} \cdot \frac{1}{2^j} = \frac{1}{2}$. The optimal makespan for this failure scenario is therefore $T_{\text{Opt}}(\mathcal{J}, \mathbf{f}) = \frac{1}{2}$.

In the above failure scenario, the maximum number of failures of any job is $f_{\max} = f_K = 2^{K-1} - 1$. Based on Lemma 9, Batch-List will complete each job J_j in $\lceil \log_2(f_j + 2) \rceil = j$ batches, and will complete all jobs in $\lceil \log_2(f_{\max} + 2) \rceil = K$ batches. Figure 2.3 illustrates the execution of this failure scenario for $K = 5$. In each batch B_k, where $1 \leq k \leq K$, the set of pending jobs is given by $\mathcal{J}_k = \{J_k, J_{k+1}, \ldots, J_K\}$. For the first batch B_1, it takes $t_1 = \frac{1}{2}$ time to complete job J_1 and thus the entire batch. For any batch B_k, where $2 \leq k \leq K - 1$, it takes $t_{k+1} = \frac{1}{2^{(k+1)}}$ time for each execution attempt of job J_{k+1}, which will have 2^{k-1} execution attempts. Thus, batch B_k will take $2^{k-1} \cdot \frac{1}{2^{(k+1)}} = \frac{1}{4}$ time to complete. The makespan of Batch-List for the entire job set $\mathcal{J}$ then satisfies:

$$\begin{aligned} T_{\text{Batch-List}}(\mathcal{J}, \mathbf{f}) &\geq \frac{1}{2} + (K-2) \cdot \frac{1}{4} \\ &= \frac{K}{4} = \frac{\lceil \log_2(f_{\max} + 2) \rceil}{2} \cdot T_{\text{Opt}}(\mathcal{J}, \mathbf{f}) \ . \end{aligned}$$

□

2.4.6 A Lower Bound on the Average-Case Performance of Batch-List

The preceding section shows that the worst-case approximation ratio of Batch-List grows linearly with the number b of batches. However, when jobs have fixed failure probabilities, the probability of having b batches tends to 0 as b approaches infinity. Thus, one might expect a constant approximation in expectation. In this section, we show that it is not true by providing an $\omega(1)$ lower bound. Despite this negative result, the experimental evaluation (in Section 2.6) shows that the average-case performance of the algorithm is very close to the optimal under many practical settings. Deriving an upper bound on the average-case approximation ratio of Batch-List remains an open question.

Theorem 10. *The expected approximation ratio of Batch-List is $\omega(1)$, if all jobs have constant failure probabilities.*

We point out that the above lower bound applies when the jobs' failure probabilities are either arbitrarily defined or related to their sequential execution times as defined in Equation (2.3). In fact, Theorem 10 holds generally true as long as the failure probability q_j of each job J_j is upper-bounded by a constant ρ, i.e., $q_j \leq \rho < 1$ for all $j = 1, \ldots, n$.

Before proving the theorem, we first compute the probability that Batch-List produces exactly b batches.

Lemma 10. *The probability that there are exactly b batches in a* Batch-List *schedule, where $b \geq 1$, is given by:*

$$Q_b = \prod_{j=1}^{n}(1 - q_j^{2^b-1}) - \prod_{j=1}^{n}(1 - q_j^{2^{b-1}-1}) \ .$$

Proof. For any $b \geq 0$, let R_b denote the probability that there are at most b batches in the schedule. According to the Batch-List algorithm, this happens when the number of failures f_j of any job J_j satisfies $f_j \leq 2^b - 2$, for all $1 \leq j \leq n$. Thus, we can compute R_b as follows:

$$\begin{aligned} R_b &= \prod_{j=1}^{n} \mathbb{P}(f_j \leq 2^b - 2) \\ &= \prod_{j=1}^{n} \sum_{k=0}^{2^b-2} \mathbb{P}(f_j = k) \\ &= \prod_{j=1}^{n} \sum_{k=0}^{2^b-2} (1 - q_j) q_j^k \\ &= \prod_{j=1}^{n} (1 - q_j^{2^b-1}) \ . \end{aligned}$$

The probability that there are exactly b batches is therefore given by $Q_b = R_b - R_{b-1}$, for any $b \geq 1$. □

(Proof of Theorem 10). To prove the claim, we show that, for any given constant $C > 0$, there exists an instance such that the expected approximation ratio of the Batch-List algorithm is strictly larger than C.

We construct the instance similarly to the one in the proof of Theorem 9. Specifically, we consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_K\}$ of K sequential jobs and at least as many processors, so that each job can be executed on a dedicated processor. For each job J_j, where $1 \leq j \leq K$, its (sequential) execution time is given by $t_j = \frac{1}{2^j}$ and its failure probability q_j is defined arbitrarily but upper-bounded by a constant $\rho < 1$.

Consider a failure scenario $\mathbf{f}$, in which each job J_j fails until batch B_{K+j} where it finally completes successfully. Hence, the total number of execution attempts of job J_j is at most 2^{K+j}, and the time to complete the job is at most $2^{K+j} \cdot \frac{1}{2^j} = 2^K$. The optimal makespan for this failure scenario therefore satisfies $T_{\text{OPT}}(\mathcal{J}, \mathbf{f}) \leq 2^K$.

Consider the Batch-List algorithm under the same failure scenario $\mathbf{f}$. In each batch B_{K+j}, where $1 \leq j \leq K-1$, job J_{j+1} does not complete successfully and is thus executed 2^{K+j-1} times. The execution time of this batch is therefore at least $2^{K+j-1} \cdot \frac{1}{2^{(j+1)}} = 2^{K-2}$. The total time to complete batches B_{K+1} to B_{2K-1}, and hence the makespan of Batch-List, is at least $T_{\text{Batch-List}}(\mathcal{J}, \mathbf{f}) \geq (K-1)2^{K-2} \geq \frac{K-1}{4} \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f})$.

Now, suppose the above failure scenario $\mathbf{f}$ happens with probability $Q(\mathbf{f}) > \frac{1}{2}$. Then, based on Equation (2.5), the expected approximation ratio of Batch-List satisfies:

$$\mathbb{E}\left[\frac{T_{\text{Batch-List}}(\mathcal{J})}{T_{\text{OPT}}(\mathcal{J})}\right] > Q(\mathbf{f}) \cdot \frac{T_{\text{Batch-List}}(\mathcal{J}, \mathbf{f})}{T_{\text{OPT}}(\mathcal{J}, \mathbf{f})} > \frac{K-1}{8}.$$

If we fix $K > 8C + 1$, we would get the results if $Q(\mathbf{f}) > \frac{1}{2}$ is true, given any bounded probabilities for the jobs. Intuitively, if a job has a very low failure probability, the probability that it completes successfully in the required batch is also very low. To resolve

this issue, we use the following technique: replace each job J_j with a cluster $\mathcal{C}_j$ of n_j jobs that are all identical to J_j, i.e., each with an execution time t_j and a failure probability q_j. We also scale up the number of processors accordingly so that each job can still be executed on a dedicated processor. Then, by choosing n_j wisely, we can make sure that cluster $\mathcal{C}_j$ completes successfully in batch B_{K+j} with high probability, and thus, collectively, the failure scenario $\mathbf{f}$ happens with high probability. To do so, we choose n_j as follows:

$$n_j = \left\lfloor \frac{2^{K+j-1} \ln(1/q_j)}{q_j^{2^{K+j-1}-1}} \right\rfloor .$$

Lemma 11. *Under the above choice of n_j and when K is large enough, the probability that any cluster $\mathcal{C}_j$, where $1 \leq j \leq K$, takes exactly $K + j$ batches to complete satisfies:*

$$S_j \geq 1 - 2^K \rho^{2^K} - \rho^{2^{K-1}} .$$

Proof. Based on Lemma 10, the probability that cluster $\mathcal{C}_j$ takes exactly $K + j$ batches to complete is given by:

$$S_j = \left(1 - q_j^{2^{K+j-1}}\right)^{n_j} - \left(1 - q_j^{2^{K+j-1}-1}\right)^{n_j} . \tag{2.12}$$

We now apply the following inequalities that hold for any $x \in [0, 1]$ and $n \in \mathbb{N}$:

$$1 - nx \leq (1 - x)^n \leq e^{-nx} .$$

In particular, the first inequality comes from the Bernoulli's Inequality, and the second inequality can be derived from the well-known inequality $(1 + 1/x)^x < e$ for any $x \geq 1$. Applying these two inequalities to Equation (2.12), we get:

$$\begin{aligned} S_j &\geq \left(1 - q_j^{2^{K+j}}\right)^{n_j} - \left(1 - q_j^{2^{K+j-1}-1}\right)^{n_j} \\ &\geq 1 - n_j q_j^{2^{K+j}} - e^{-n_j q_j^{2^{K+j-1}-1}} . \end{aligned} \tag{2.13}$$

We will now provide upper bounds for the second term $X_j = n_j q_j^{2^{K+j}}$ and the third term $Y_j = e^{-n_j q_j^{2^{K+j-1}-1}}$.

To bound X_j, we note that $\ln(x) \leq x$ for any $x > 0$ and $n_j \leq \frac{2^{K+j-1} \ln(1/q_j)}{q_j^{2^{K+j-1}-1}}$. The second term then satisfies:

$$\begin{aligned} X_j &\leq \frac{2^{K+j-1} \ln(1/q_j)}{q_j^{2^{K+j-1}-1}} q_j^{2^{K+j}} \\ &\leq \frac{2^{K+j-1}}{q_j^{2^{K+j-1}}} q_j^{2^{K+j}} \\ &= 2^{K+j-1} q_j^{2^{K+j-1}} \\ &\leq 2^{K+j-1} \rho^{2^{K+j-1}} . \end{aligned}$$

Further, we can easily check that $x\rho^x$ is a decreasing function of x when $x \ln(\rho) < -1$. Thus, when K is large enough, and since $j \geq 1$, we have $2^{K+j-1} \rho^{2^{K+j-1}} \leq 2^K \rho^{2^K}$. Therefore, we can get the following upper bound for the second term:

$$X_j \leq 2^K \rho^{2^K} . \tag{2.14}$$

To bound the third term, we note that $n_j \geq \frac{2^{K+j-1}\ln(1/q_j)}{2 \cdot q_j^{2^{K+j-1}-1}}$, so we can get:

$$\begin{aligned} Y_j &\leq e^{-\frac{2^{K+j-1}\ln(1/q_j)}{2 \cdot q_j^{2^{K+j-1}-1}} q_j^{2^{K+j-1}-1}} \\ &= e^{-2^{K+j-2}\ln(1/q_j)} \\ &= q_j^{2^{K+j-2}} \\ &\leq \rho^{2^{K+j-2}} \\ &\leq \rho^{2^{K-1}} \quad . \end{aligned} \tag{2.15}$$

The lemma is then proved by substituting Inequalities (2.14) and (2.15) into Inequality (2.13). □

Based on the result of Lemma 11, the probability of the desired failure scenario $\mathbf{f}$ can be computed as:

$$\begin{aligned} Q(\mathbf{f}) &= \prod_{j=1}^{K} S_j \\ &\geq \left(1 - 2^K \rho^{2^K} - \rho^{2^{K-1}}\right)^K \\ &\geq 1 - K2^K \rho^{2^K} - K\rho^{2^{K-1}} \quad . \end{aligned}$$

The last inequality is again due to the Bernoulli's Inequality.

For any constant $\rho < 1$, two terms above, namely, $K2^K\rho^{2^K}$ and $K\rho^{2^{K-1}}$ both tend to 0 as $K \to \infty$. Therefore, there must exist a K^* such that $Q(\mathbf{f}) \geq \frac{1}{2}$. Setting $K = \max(K^*, 8C+1)$ proves the theorem. □

2.5 A Lower Bound of Any Algorithm for Arbitrary Speedup Model

So far, we have focused on special speedup models. In this section, we show that the competitive ratio of any deterministic online algorithm (including ours) can be unbounded under an *arbitrary* speedup model[7].

Theorem 11. *Any algorithm is at least $\Omega(\ln(f_{max}))$-competitive under an arbitrary speedup model.*

Proof. We fix an arbitrary integer $\ell > 1$ and set $K = 2^\ell$. The instance consists of $n = 2^K - 1$ identical tasks organized in groups. Specifically, for any $i \in [1, K]$, group i contains 2^{K-i} tasks, each with exactly $i-1$ failures. Thus, $f_{max} = K - 1$. Figure 2.4 shows such an instance for $\ell = 2, K = 4$ and $n = 15$. All tasks in the graph are identical, with an execution time function $t(p) = \frac{1}{\log_2(p)+1}$. We set the total number of processors to be $P = K \cdot 2^{K-1}$.

We show that the optimal offline algorithm completes the above instance with a makespan at most 1, whereas any deterministic online algorithm may produce a makespan at least $\ln(K) - \ln(\ell) - \frac{1}{\ell}$, thus proving the result.

First, the optimal offline algorithm could schedule the tasks as follows: for any group $i \in [1, K]$, it allocates 2^{i-1} processors to each linear chain in the group. The total number of

[7]Under an arbitrary speedup model, the execution time $t(p)$ of a task can take any arbitrary function of its processor allocation p.

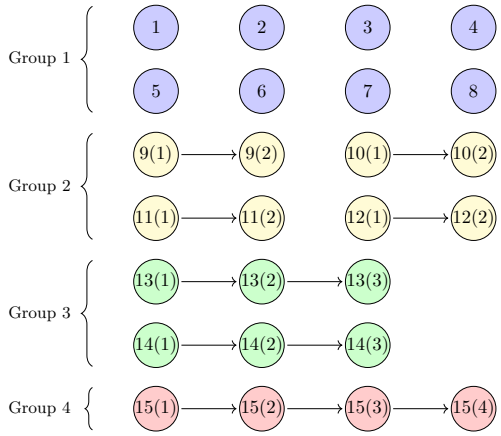


Figure 2.4: A lower bound instance in Theorem 11 with $\ell = 2$, $K = 4$, and $n = 15$ linear task chains. Each circle represents a task and the number inside each circle indicates the ID of the linear chain the task is in (and the number in the parenthesis indicates the task's position in that linear chain).

required processors is then $\sum_{i=1}^{K} 2^{i-1} \times 2^{K-i} = K \times 2^{K-1} = P$. Thus, all linear chains could be executed in parallel. Furthermore, they will all be completed at time 1, since each task in group i has i repetitions, and each task has an execution time $t(2^{i-1}) = \frac{1}{\lg(2^{i-1})+1} = \frac{1}{i}$. Figure 2.5(a) illustrates the schedule for this instance with $\ell = 2$.

Now, we establish a lower bound on the makespan of any deterministic online algorithm. For any $i \in [1, K-1]$, let L_i denote the set of tasks in all groups $j \leq i$, and let L'_i denote the set of tasks in all groups $j > i$. Let us define t_i to be the first time a task in L'_i completes i repetitions. We further define $t_0 = 0$ and let t_K denote the makespan of the online algorithm.

Lemma 12. *Any algorithm could produce a schedule that satisfies $t_i - t_{i-1} \geq \frac{1}{\ell+i}$, for all $i \in [1, K]$.*

Proof. Since all tasks are identical, an online algorithm cannot distinguish them. Thus, for any $i \in [1, K]$, an adversary could make tasks that first complete i repetitions by the online algorithm be from L_i. Therefore, at time t_i, all tasks containing exactly i repetitions (i.e., the ones from group i) are already completed, and at time t_{i-1}, no task has started its i-th repetition by definition (this also holds for t_0 and t_K). Hence, all repetitions in the i-th position of the tasks in group i must be entirely processed between t_i and t_{i-1}, and the number of such tasks is 2^{K-i}.

For the sake of contradiction, suppose we have $t_i - t_{i-1} < \frac{1}{\ell+i}$. Thus, the execution time of these tasks must satisfy $t(p) = \frac{1}{\lg(p)+1} \leq \frac{1}{\ell+i}$, hence their processor allocation must be at least $p \geq 2^{\ell+i-1} = K \cdot 2^{i-1}$. As the area of the task $a(p) = pt(p) = \frac{p}{\lg(p)+1}$ is increasing with the number of processors, the total area of all tasks that needs to be processed between t_i and t_{i-1} is at least $2^{K-i} \cdot a(K \times 2^{i-1}) = \frac{2^{K-i} \cdot K \cdot 2^{i-1}}{\lg(K \cdot 2^{i-1})+1} = \frac{K \cdot 2^{K-1}}{\ell+i} = \frac{P}{\ell+i}$. Since we have P processors, the total time required to process this area is at least $\frac{1}{\ell+i}$, which contradicts $t_i - t_{i-1} < \frac{1}{\ell+i}$. □

One strategy to cope with the worst-case scenario above is to allocate the same number of processors to each linear chain (or more precisely allocate one more processor to some

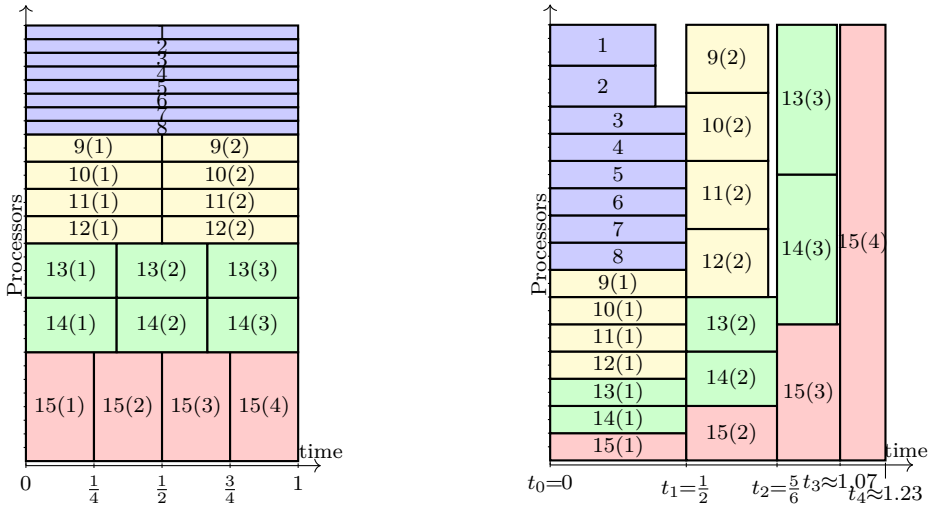


Figure 2.5: For the lower bound instance of Figure 2.4: (a) An offline schedule with a makespan of 1; (b) An online algorithm's schedule, allocating (approximately) the same number of processors to all linear chains and producing a makespan of $t_4 \approx 1.23$.

linear chains in order to utilize all the processors). Figure 2.5(b) illustrates this strategy for the same instance with $\ell = 2$. This is roughly what Batch-List does.

Finally, we can use the result of Lemma 12 to lower bound the makespan of an online algorithm, which is given by $t_K = \sum_{i=1}^{K}(t_i - t_{i-1})$. Since for all j, $\ln(j) + \gamma < \sum_{i=1}^{j} \frac{1}{i} < \ln(j) + \gamma + \frac{1}{j}$ where γ is the Euler constant, we obtain:

$$
\begin{aligned}
t_K &\geq \sum_{i=1}^{K} \frac{1}{\ell + i} > \sum_{i=\ell+1}^{K} \frac{1}{i} = \sum_{i=1}^{K} \frac{1}{i} - \sum_{i=1}^{\ell} \frac{1}{i} \\
&> (\ln(K) + \gamma) - \left(\ln(\ell) + \gamma + \frac{1}{\ell}\right) = \ln(K) - \ln(\ell) - \frac{1}{\ell} \,. \qquad \square
\end{aligned}
$$

Remark 1. *This shows that the competitive ratio of Batch-List is on the same order of magnitude as the best competitive ratio achievable.*

2.6 Performance Evaluation

In this section, we evaluate and compare the performance of different scheduling algorithms using simulations on synthetic moldable jobs that follow various speedup models.

2.6.1 Simulation Setup

Evaluated Algorithms: We evaluate the performance of our two scheduling algorithms, namely, Lpa-List (or Lpa in short) and Batch-List (or Batch in short). For Batch, we set $\epsilon = 0.3$ for its processor allocation procedure (Lemma 7). Their performance is also compared against that of the following two baseline heuristics:

- MinTime: allocates processors to minimize the execution time of each job and schedules all jobs using the List strategy (Algorithm 1). This is also known as the shortest execution time (Set) algorithm in [78];
- MinArea: allocates processors to minimize the area of each job and schedules all jobs using the List strategy.

Priority Rules: We consider three priority rules that have been shown to give good performance when (rigid) jobs are scheduled with the List strategy [25], which is used in all four evaluated algorithms (recall that Batch uses List in each batch). The three priority rules are:

- LPT (Longest Processing Time): a job with a longer processing time has a higher priority;
- HPA (Highest Processor Allocation): a job with a higher processor allocation has a higher priority;
- LA (Largest Area): a job with a larger area has a higher priority.

Speedup Models: We generate synthetic moldable jobs that follow six speedup models: roofline, communication, Amdahl, mix (in two different versions) and power. Each job J_j is defined by two parameters: the total work w_j (i.e., the sequential execution time), which is drawn uniformly in $[5000, 4000000]$, and another parameter that depends on the speedup model.

- **Roofline**: the maximum degree of parallelism $\bar{p}_j$ is an integer drawn uniformly in $[100, 4000]$;
- **Communication**: the communication overhead is set as $c_j = \alpha \cdot 2^r$, where r is an integer uniformly chosen in $[0, 3]$ and α is drawn uniformly in $[1, 2]$.
- **Amdahl**: the sequential fraction is set as $\gamma_j = \frac{\alpha}{10^r}$, where r is an integer uniformly chosen in $[2, 7]$ and α is drawn uniformly in $[0, 10]$.
- **Mix**: we consider two different parameter settings: the first one, called **mix-low-com**, uses the same set of parameters as what is chosen for the roofline, communication, and Amdahl's model. The second one, called **mix**, uses $3c_j$ instead of c_j for the communication overhead.
- **Power**: the parameter δ_j is chosen uniformly in $[0, 1]$.

Failure Distribution: To generate failures for the jobs, we assume that silent errors follow the exponential distribution [80]. Let λ denote the error rate per unit of work, so a job will be struck by a silent error for every $1/\lambda$ unit of work executed on average. Following our failure model (Section 2.3), we assume parallelizing a job does not change the total number of computational operations (it may increase the communication, which we consider protected). Hence, the failure probability of a job will not depend on its processor allocation nor its execution time, but solely on its total work. For a job J_j with total work w_j, its failure probability is given by $q_j = 1 - e^{-\lambda w_j}$.

In the simulations, we set $\lambda = 10^{-7}$ by default. Given the chosen values of w_j, this corresponds to a failure probability between 0.0005 and 0.33 for a job. We also set the default number of processors and number of jobs to be $P = 7500$ and $n = 500$, but we will also vary all of these parameters to evaluate their impact on the performance.

Evaluation Methodology: The evaluation is done as follows: we generate 30 different sets of jobs, and for each set, 100 failure scenarios are drawn randomly from the failure distribution described above. For each of the failure scenarios, the simulated makespan of an algorithm is normalized by a lower bound (described below), which is then averaged over the 100 failure scenarios to estimate the expected ratio for the job set. Lastly, this ratio is averaged over the 30 job sets to compute the final expected performance of the algorithm. In addition, we also estimate the worst-case performance of the algorithm by using its largest normalized makespan over all job sets and failure scenarios.

Given job set $\mathcal{J}$ and a failure scenario $\mathbf{f}$, the makespan lower bound given in Equation (2.8) depends on the processor allocation and hence the scheduling algorithm. To ensure that the performance of all algorithms is normalized by the same quantity, we use the following rather loose lower bound, which is, however, independent of the scheduling

decision:

$$L'(\mathcal{J}, \mathbf{f}) = \max\left(t'_{\max}(\mathcal{J}, \mathbf{f}), \frac{A'(\mathcal{J}, \mathbf{f})}{P}\right),$$

where $t'_{\max}(\mathcal{J}, \mathbf{f}) = \max_j \min_p (f_j + 1) t_j(p)$ is the minimum possible maximum execution time of all jobs, and $A'(\mathcal{J}, \mathbf{f}) = \sum_j \min_p (f_j + 1) a_j(p)$ is the minimum possible total area. Since this lower bound gives a pessimistic estimation on the optimal schedule, the actual performance of the algorithms is likely to be better than reported.

The simulation code for all experiments is publicly available at http://www.github.com/vlefevre/job-scheduling.

2.6.2 Comparison of Algorithms and Priority Rules

We first compare the performance of different algorithms and study the impact of priority rules on their performance.

Figure 2.6 shows the normalized makespans for the 11 combinations of algorithms and priority rules under all speedup models. For the MinArea algorithm, priority rules LA and LPT are identical, as the algorithm allocates one processor to all jobs, so only the results of LPT are reported. As we can see, MinArea fares poorly in most cases, because it allocates one processor to each job in order to minimize the area. This results in very long job execution (and re-execution) times, which leads to extremely large makespan. Moreover, allocating only one processor per job also results in idle processors thus resource inefficiency whenever the number of processors is higher than the number of jobs. The MinTime algorithm performs well for the roofline and mix models, but as more overhead is introduced in the communication, Amdahl and power models, it continues to allocate a large number of processors to the jobs in order to minimize the execution time. This leads to a significant increase in the total area and hence degrades the performance. On the other hand, the Lpa and Batch algorithms maintain a good balance between the execution time and area of a job, thus they perform well for all speedup models in terms of both expected performance (bars) and worst-case performance (top endpoints of lines). Independently of the priority rules, Lpa performs the best for the roofline and communication models while Batch performs the best for the other models.

Figure 2.7 further shows the results of four combinations of P and n with similar performance trends. We notice that these two parameters do have an impact on the performance of Batch under the communication, Amdahl and mix models, in particular at $P = 1000$ and $n = 500$. Indeed, under these models and when P is significantly larger than n, Batch tends to reduce all jobs to similar length and execute them at the same time, which gives the best tradeoff between the area and maximum execution time. In that case, the first batch, where all jobs are executed exactly once, is done almost perfectly. As the makespan of the first batch is dominant under $\lambda = 10^{-7}$, the overall makespan is closer to the lower bound. However, with $P = 1000$ and $n = 500$, there are not enough processors to execute all jobs at the same time. Thus, the performance of Batch becomes worse than that of Lpa.

We also notice that the performance of MinTime under the two mix models becomes better when the number of processors is large compared to the number of jobs (e.g., $P = 10000, n = 100$). Indeed, MinTime is able to simultaneously minimize the execution time of all jobs in this case without using up all the processors, thus achieving near-optimal performance. Note this is not possible with fewer processors, as minimizing the execution time alone for each job will increase the total area, which also plays an important role under such circumstance to have overall good performance.

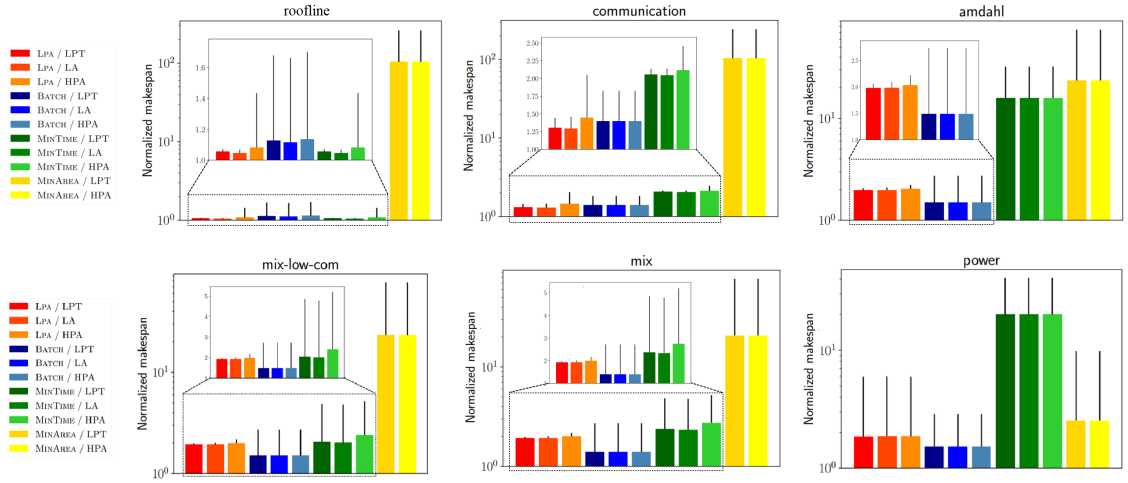


Figure 2.6: Performance of different algorithms and priority rules under six speedup models with $P = 7500$, $n = 500$ and $\lambda = 10^{-7}$. The bars represent expected performance and the top endpoints of the lines represent worst-case performance.

Comparing the three priority rules, no significant difference is observed. In general, LPT and LA give similar results, and slightly better results than HPA. This is consistent with the results observed in [25] for scheduling rigid jobs. Given these results, we will only consider the LPT priority rule in the subsequent evaluation. We will also omit the MINAREA and MINTIME algorithms for the models under which they perform badly, while focusing on comparing the expected performance of the remaining algorithms.

2.6.3 Impact of Different Parameters

We now study the impact of different parameters on the performance of the algorithms. We start from $P = 7500$, $n = 500$, and $\lambda = 10^{-7}$, and vary one of these parameters in each experiment.

Impact of Number of Processors (P): Figure 2.8 shows the performance when the number of processors P is varied between 1000 and 15000 for different speedup models. For the roofline model, all three algorithms return the same processor allocation, i.e., the maximum degree of parallelism or the maximum number of processors, for each job. Further, both LPA and MINTIME use the LIST strategy for scheduling, so the two algorithms have exactly the same performance. In contrast, BATCH does not perform as well, because it schedules the jobs in batches, and thus needs to wait for every job in a batch to finish before starting the next one, which causes delays. The initial up-and-down of the normalized makespans is due to the upper limit (i.e., 4000) we set on the maximum degree of parallelism: when $P \ll 4000$, few processors are wasted so the resulting schedules are very efficient; when $P \gg 4000$, most jobs are fully parallelized and thus completed faster. For BATCH, however, the proportion of idle processors at the end of a batch increases with P, which explains the widening of performance gap from the other two algorithms.

For the communication model, parallelizing a job becomes less efficient due to the extra communication overhead, so BATCH starts to perform better than MINTIME thanks to its smarter processor allocation strategy. Here, both BATCH and LPA have similar processor allocations, so the performance difference between the two algorithms is still induced by the idle times at the end of the batches, which are again increasing with the number of processors.

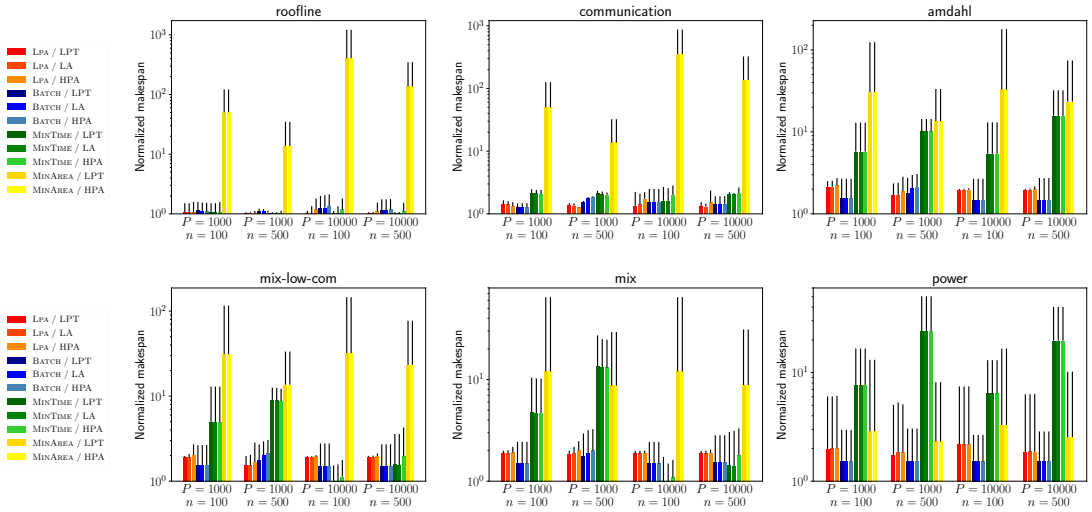


Figure 2.7: Performance of different algorithms and priority rules under six speedup models with $\lambda = 10^{-7}$ and four other different combinations of P and n. The bars represent expected performance and the top endpoints of the lines represent worst-case performance.

For the Amdahl's model, the results look very different, as BATCH now outperforms LPA despite the idle time at the end of each batch. This is due to BATCH's ability to better balance the job execution times globally, which becomes more important in this case. Moreover, the trend is not affected by the number of processors.

For the two mix models, LPA and BATCH behave similarly as in the Amdahl's model, because they tend to allocate a relatively small number of processors for each job, thus the maximum degree of parallelism is not reached and the communication cost is relatively small. We also notice that the performance of MINTIME is getting better with increasing number of processors, especially under higher communication cost. Indeed, contrary to the Amdahl's model (where the execution time of a job is minimized when we allocate all the processors), the minimum execution time of a job is achieved with a reasonable number of processors because of the communication overhead. Thus, when P is high enough such that all jobs can be processed in parallel while minimizing their execution times, MINTIME's allocation becomes close to optimal.

Unlike the previous models, the power model has a relatively slow-increasing speedup curve, thus allocating one processor to each job as in MINAREA is not a bad choice. For the same reason, MINTIME that allocates all the processors to a job performs badly, so it is not showed here. The relative performance of LPA and BATCH is similar to that in the Amdahl's and mix models, again due to BATCH's coordinated processor allocation strategy. Because of the jobs' slow speedup curves, the benefit of allocating more processors also gets smaller, thus having more processors barely impacts the performance of the algorithms.

Impact of Number of Jobs (n): Figure 2.9 shows the performance when the number of jobs n is varied between 100 and 1000. Again, we can see that BATCH performs the worst in the roofline model, gets better than MINTIME in the communication model, and has the best performance in the other models. While the varying number of jobs has a small impact on the performance of LPA, the performance of BATCH improves as the number of jobs increases in the roofline and communication models. Indeed, with a constant number of processors P, having more jobs decreases the number of available processors per job,

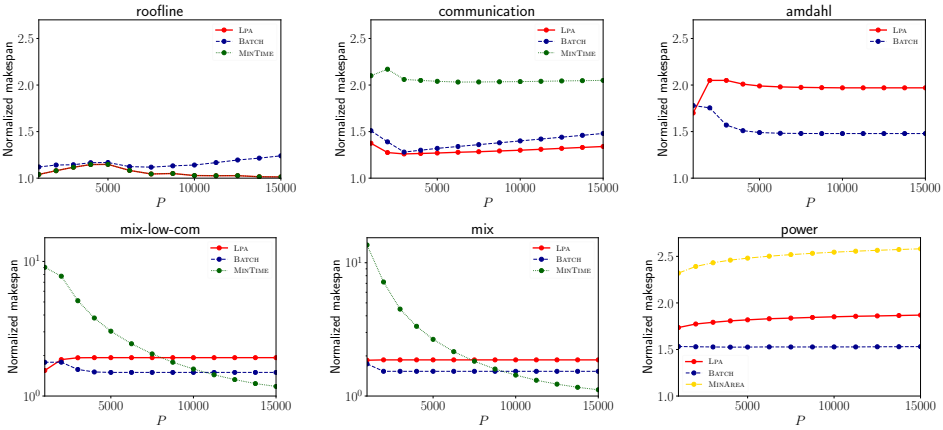


Figure 2.8: Performance of the algorithms for different speedup models with $n = 500$, $\lambda = 10^{-7}$ and $P \in [1000, 15000]$.

thus reduces the performance gap between scheduling algorithms due to the idle processors between batches. For the other models, the number of jobs has a small impact even for BATCH. Overall, as the number of jobs increases, the trend in the relative performance of the algorithms is consistent with the previous results we have observed in Figure 2.8 when the number of processors decreases.

Impact of Error Rate (λ): Figure 2.10 shows the impact of the error rate λ when it is varied between 10^{-8} (corresponding to 0.03 error per job on average) and 10^{-6} (corresponding to 12 errors per job on average). Once again, the relative performance of the three algorithms remains the same as before under the respective speedup models. While the performance of LPA is barely affected, which is not surprising considering that its processor allocation is performed locally and separately from job scheduling, the performance of BATCH gets worse with increasing error rate λ (and hence the number of failures), which corroborates the theoretical analysis (Theorem 8). In particular, when the error rate is small, there are very few failures and almost all jobs will complete in one batch. In this case, the processor allocation procedure of BATCH (Lemma 7) is very precise. With increased error rate, more failures will occur and thus more batches will be introduced, causing scheduling inefficiencies from both idle times between the batches and possible imprecision in the processor allocations (especially with a large batch, since the actual number of failures may deviate significantly from the anticipated values). Finally, although the processor allocation is also performed locally for MINTIME and MINAREA, the effect of increasing λ is similar to that of increasing P (or the opposite to that of increasing n): when there are more failures, we spend more time processing few large jobs that fail a lot, meaning that after some time only very few jobs are not finished yet. This effectively increases the total number of processors for these jobs or reduces the total number of jobs.

2.6.4 Summary of Results

Table II summarizes the makespan ratios of the four algorithms over the entire set of experiments, in terms of both average-case performance (expected ratio) and worst-case performance (maximum ratio). Overall, the results confirm the efficiency of our two resilient scheduling algorithms (LPA and BATCH), which outperform the baseline heuristics (MINTIME and MINAREA) in all settings. For the simplest roofline model, LPA is equivalent to MINTIME, both achieving a makespan very close to the lower bound (with a ratio around 1.06 on average). For the other models, we can observe significant performance difference

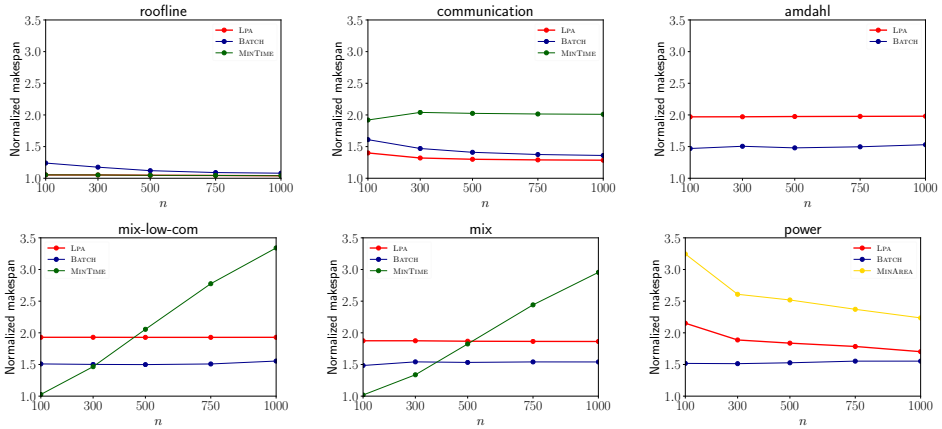


Figure 2.9: Performance of the algorithms for different speedup models with $P = 7500$, $\lambda = 10^{-7}$ and $n \in [100, 1000]$.

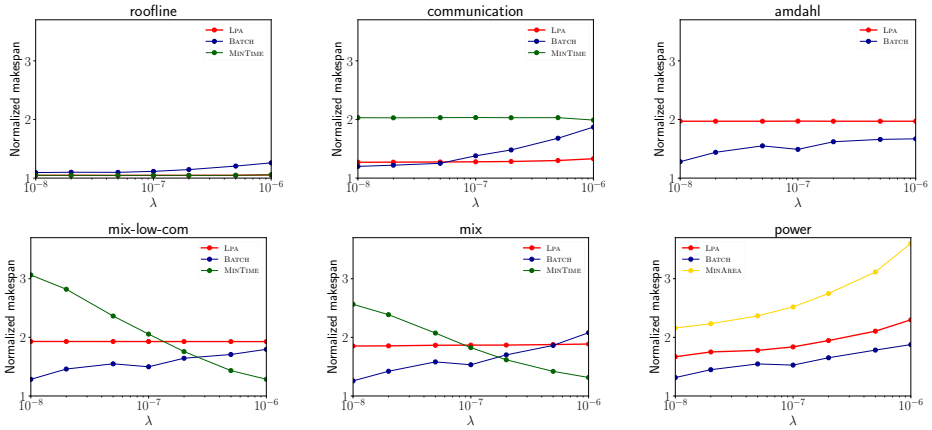


Figure 2.10: Performance of the algorithms for different speedup models with $P = 7500$, $n = 500$ and $\lambda \in [10^{-8}, 10^{-6}]$.

Table II: Summary of the performance for the four algorithms (with LPT priority rule) under the six speedup models.

Speedup Model		Roofline	Communication	Amdahl	Mix-low-com	Mix	Power
LPA	Expected	1.057	1.312	1.961	1.896	1.867	1.861
	Maximum	1.219	2.241	2.349	1.987	1.995	9.655
BATCH	Expected	1.158	1.434	1.529	1.548	1.571	1.549
	Maximum	1.999	2.449	2.874	3.674	4.164	3.975
MINTIME	Expected	1.057	2.044	15.567	2.810	2.704	20.386
	Maximum	1.219	2.666	49.795	12.611	27.174	61.726
MINAREA	Expected	114.079	122.199	23.594	16.875	9.686	2.571
	Maximum	1217.13	871.38	199.572	259.163	120.9	27.109

between our best algorithm and the baseline. In particular, LPA achieves good performance with an expected ratio around 1.3 for the communication model, and an expected ratio less than 2 for the other models. We also notice that the maximum ratios are only slightly larger than the ones in the average case, and they remain much lower than those predicted by the theoretical bounds (except for the power model where the ratio is more than 9). BATCH also achieves excellent results thanks to its coordinated processor allocation and failure handling ability. It achieves a better average ratio (less than 1.6) for all models, but has larger worst-case ratios compared to LPA (except for the power model). On the other hand, the two baseline heuristics, although doing well in some scenarios, tend to have more irregular performance that depends on the model and parameter. In contrast, our algorithms exhibit more robust performance under various models and parameter settings.

2.7 Conclusion and Future Work

In this chapter, we have studied the problem of scheduling moldable parallel jobs to cope with silent errors.

We present a formal model of the problem and design two resilient scheduling algorithms (LPA and BATCH). While not knowing the failure scenarios of the jobs in advance, LPA utilizes a delicate local processor allocation strategy and BATCH extends the notion of batches to coordinate the processor allocations. Both algorithms use an extended LIST strategy with failure-handling ability to schedule the jobs. On the theoretical side, we derived new approximation results for both algorithms under several classical speedup models. In particular, LPA is shown to be a constant approximation for the roofline model, the communication model, the Amdahl's model, as well as a mixed model. We also derived its approximation ratios for the power model and general monotonic model.

Also, we prove that BATCH achieves a $\Theta(\log_2 f_{\max})$-approximation for arbitrary speedup models, where $f_{\max}$ is the maximum number of failures of any job in a failure scenario. All of these results are worst-case results: they hold for any failure scenario. We also derived an $\omega(1)$ lower bound on the average-case performance of BATCH. Extensive simulations show good performance of the two proposed algorithms compared to some baseline heuristics, demonstrating their practical usefulness and robustness under common job speedups and parameter settings.

Future work will be devoted to the investigation of alternative failure models, such as fail-stop errors (as opposed to silent errors) or schedule-dependent failure probabilities (that depend on the number of processors allocated to a job, and hence on its area).

One may also consider checkpointing and rollback recovery for long-running jobs to avoid re-executing a failed job from scratch. On the practical side, we seek to validate the performance of our algorithms by evaluating them using datasets extracted from job execution logs with realistic speedup profiles and failure traces.

Chapter 3

Online Scheduling of Moldable Task Graphs under Common Speedup Models

In Chapter 2, we have studied the scheduling of moldable tasks subject to failures, where an instance could be seen as a graph of linear chains of identical tasks. In each chain, only the first task is *visible*, i.e., we only discover a task when its predecessor is completed. In this chapter, we extend this study to the general online scheduling problem graph of moldable task graphs on multiprocessor systems, where we still aim at minimizing the overall completion time (or makespan). Moldable job scheduling has been widely studied in the literature, in particular when tasks have dependencies (i.e., task graphs) or when tasks are released on-the-fly (i.e., online). However, few studies have focused on both (i.e., online scheduling of moldable task graphs). In this chapter, we design a new online scheduling algorithm for this problem and derive constant competitive ratios under several common yet realistic speedup models (i.e., roofline, communication, Amdahl, and a general combination). We also prove, for each speedup model, a lower bound on the competitiveness of any online list scheduling algorithm that allocates processors to a task based only on the task's parameters and not on its position in the graph. This lower bound matches exactly the competitive ratio of our algorithm for the roofline, communication and Amdahl's model, and is close to the ratio for the mix model. Finally, we provide a lower bound on the competitive ratio of any deterministic online algorithm for the arbitrary speedup model, which is not constant but depends on the number of tasks in the longest path of the graph. This chapter corresponds to Submission [S5] (see Chapter 9). Although Publication [C3] focused on the same problem, the analysis was sharpened significantly since then, with strong additional results on lower bounds. Therefore, the initial algorithm and analysis from Publication [C3] are obsolete and omitted.

3.1 Introduction

This work investigates the online scheduling of parallel task graphs on a set of identical processors, where each task in the graph is *moldable*. In the scheduling literature, a moldable task (or job) is a parallel task that can be executed on an arbitrary but fixed number of processors. The execution time of the task depends upon the number of processors used to execute it, which is chosen once and for all when the task starts its execution but cannot be modified later on during execution. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (*rigid* tasks) and to a variable dynamic allocation (*malleable* tasks) [59].

Moldable tasks offer a nice trade-off between rigid and malleable tasks: they easily

n	Number of tasks
P	Number of processors
μ	Maximum fraction of processors used by our algorithm
Opt	Optimal Offline Scheduler
w_j	Work of a given task (sequential execution time)
d_j	Inherently sequential fraction of a task (cannot be parallelized)
c_j	Communication overhead per processor for a task
$\bar{p}_j$	Maximum degree of parallelism of a task
$t_j(p_j)$	Execution time of task j when allocated p_j processors
Roo	Roofline Model
Com	Communication Model
Amd	Amdahl Model
Mix	Mix Model

Table I: Summary of main notations for Chapter 3.

adapt to the number of available resources, contrarily to rigid tasks, while being easy to design and implement, contrarily to malleable tasks. Thus, many computational kernels in scientific libraries (e.g., for numerical linear algebra and tensor computations) can be deployed as moldable tasks to efficiently run on a wide range of processors. Because of the importance and wide availability of moldable tasks, scheduling algorithms for such tasks have received considerable attention in the literature (see Section 3.2 for details). Like many other scheduling problems, scheduling moldable tasks comes in different flavors, and the following provides a brief taxonomy:

- **Offline Scheduling vs. Online Scheduling.** In the offline version of the scheduling problem, all tasks are known in advance, before the execution starts. The problem is $\mathcal{NP}$-complete for both independent and dependent tasks [160], and the goal is to design scheduling algorithms with good *approximation ratio*, which measures the worst-case performance of an algorithm against an optimal scheduler for all possible input instances. On the contrary, in the online version of the scheduling problem, tasks are released on the fly, and the objective is to design online algorithms with good *competitive ratio* [144], against an optimal offline scheduler that knows in advance all the tasks and their dependencies in the graph. The competitive ratio is established against all possible strategies devised by an adversary trying to force the online algorithm to take *bad* decisions.
- **Independent Tasks vs. Task Graphs.** In a scheduling problem, different tasks can be either independent of each other or dependent forming a task graph. If tasks are independent, they are either all known to the scheduling algorithm initially (in the offline version), or released on the fly and the scheduler only discovers their characteristics upon release (in the online version). For task graphs, either the entire graph is known at the start (in the offline version), or each new task along with its characteristics is only released when all of its predecessors have completed execution (in the online version). For the latter case, the shape of the graph as well as the nature of the tasks are not known in advance, and they are revealed only as the execution progresses.

In this chapter, we investigate arguably the most difficult version of the problem, namely, the *online scheduling of moldable task graphs*, which is the version that has received the least research attention so far. The objective is to minimize the overall completion time of the task graph, or the *makespan*. We assume that the scheduling of each task is

non-preemptive and without restarts [60], which is highly desirable to avoid large overheads incurred by checkpointing partial results, context switching, and task migration, etc.

While the performance of a scheduling algorithm greatly depends upon the speedup model of the tasks, we consider several common yet realistic speedup models, including the roofline model, the communication model, the Amdahl's model, and a general combination of them (as already seen in Chapter 2, and see Section 3.3.1 for their precise definitions). These models have been widely assumed and studied in the literature for modeling the scaling behavior of parallel applications. We design a new online scheduling algorithm that achieves good competitive ratios for these models. This is done through a novel analysis framework, which provides a tighter and more coupled analysis between a local processor allocation algorithm and the list scheduling algorithm. To the best of our knowledge, a competitive ratio was previously known only for task graphs under the roofline model [60], while our work offers the first results for several other speedup models, and these results lay the foundations for this scheduling problem.

Our main contributions are summarized as follows:

- We present a new online algorithm and prove its constant competitive ratio for the four speedup models. In particular, results are derived for the communication model, the Amdahl's model, and the mix model.
- For each speedup model, we prove a lower bound on the competitiveness of any online list scheduling algorithm whose processor allocation is *local and deterministic*, i.e., the decision depends only on P and the task's parameters, but not on its position in the graph. The results show that our algorithm achieves the best possible competitive ratios for the roofline, communication and Amdahl's models for this class of algorithms.

The rest of this chapter is organized as follows. Section 3.2 surveys some related works on moldable task scheduling. The formal model and problem statement are presented in Section 3.3. Section 3.4 introduces the new online algorithm and proves its competitive ratios for different speedup models. Section 3.5 presents, for each model, a lower bound of any online list scheduling algorithm with deterministic local processor allocation. Our algorithm belongs to this class and has the best possible competitive ratio for the roofline, communication, and Amdahl's models. Finally, Section 3.6 concludes the chapter and provides hints for future directions.

3.2 Related Work

In this section, we discuss some related works on moldable task scheduling. Following the taxonomy of the previous section, we consider four versions of the problem combining offline vs. online scheduling and independent tasks vs. task graphs scheduling. We mainly focus on works that have derived approximation or competitive ratios. While some of these results depend on specific speedup models, others hold for a more general class of models.

Offline Scheduling of Independent Tasks

Belkhale and Banerjee [16] considered moldable tasks that follow the monotonic model, where the execution time of a task is non-increasing and the area (processor allocation times execution time) is non-decreasing with the number of processors. They presented a $\frac{2}{1+1/P}$-approximation algorithm by iteratively updating the processor allocations. For the same model, Błażewicz et al. [29] also presented a 2-approximation algorithm while relying on an optimal continuous schedule. Mounié et al. [123] presented a $(\sqrt{3}+\epsilon)$-approximation

algorithm using dual approximation. They later improved the ratio to $1.5+\epsilon$ [124]. Finally, Jansen and Land [90] proposed a Polynomial-Time Approximation Scheme (PTAS) for the problem.

If the execution time of a task can be an arbitrary function of the processor allocation (i.e., the arbitrary model), Turek et al. [159] designed a 2-approximation list-based algorithm and a 3-approximation shelf-based algorithm. Ludwig and Tiwari [116] showed the same result but with lower computational complexity. When each task only admits a subset of all possible processor allocations, Jansen [89] presented a $(1.5+\epsilon)$-approximation algorithm, which is tight since the problem cannot have an approximation ratio better than 1.5 unless $\mathcal{P} = \mathcal{NP}$ [96]. When the number of processors is a constant or polynomially bounded by the number of tasks, Jansen et al. [92] showed that a PTAS exists.

Offline Scheduling of Task Graphs

For offline scheduling of moldable task graphs, Wang and Cheng [163] showed that the earliest completion time algorithm is a $(3 - \frac{2}{P})$-approximation for the roofline model. Since the processor allocation is done independently for each task, their algorithm and corresponding ratio can also be applied to the online setting as discussed below.

For the monotonic model, Belkhale and Banerjee [17] presented a 2.618-approximation algorithm while assuming the availability of an optimal processor allocation. Lepère et al. [110] proposed an algorithm with an approximation ratio of 5.236. They also showed that the optimal allocation can be achieved in pseudo-polynomial time for some special graphs, such as series-parallel graphs and trees, thus leading to a 2.618-approximation for these graphs. Jansen and Zhang [94] later improved the approximation ratio for general graphs to around 4.73. When assuming that the area of a job is a concave function of the number of processors, Jansen and Zhang [93] proposed a 3.29-approximation algorithm. Chen and Chu [42] improved the ratio to around 2.95 by further assuming that the execution time of a job is strictly decreasing in the number of allocated processors.

Online Scheduling of Task Graphs

Feldmann et al. [60] designed an online algorithm to schedule moldable task graphs under the roofline model. They showed that their algorithm achieves a competitive ratio of 2.618, thus improving the previous result by Wang and Cheng [163]. Furthermore, their algorithm works in the *non-clairvoyant* setting, where the task execution time is also unknown to the scheduler. In the last chapter we investigated the problem of scheduling moldable tasks subject to failures, where a task needs to be re-executed after a failure until it is successfully completed. This corresponds to a special task graph consisting of multiple linear chains (one per task), where the length of each chain corresponds to the total number of executions of a task. The problem is semi-online, since all the tasks are known at the beginning, but the task failures and hence their re-executions are only discovered on-the-fly. They considered several common speedup models (as in this chapter) and presented a scheduling algorithm that achieves constant competitive ratios for these models. In this chapter, we study the general online scheduling of moldable task graphs (as in [60]). We present improved competitive ratios as well as lower bounds for the common speedup models. We do not consider task failures as before, but our results can readily carry over to the failure scenario.

Table II summarizes different versions of the moldable task scheduling problem together with the related papers within each version.

Table II: Different versions of the moldable task scheduling problem, and related papers in each version.

	Offline scheduling	**Online scheduling**
Independent tasks	*Monotonic model*: [16, 29, 90, 123, 124] *Arbitrary model*: [89, 90, 116, 159]	*Comm. model (over time)* [54, 78, 102] *Arbitrary model (one-by-one)*: [170]
Task graphs	*Roofline model*: [163] *Monotonic model*: [17, 42, 93, 94, 110]	*Roofline model*: [60, 163] *Common models*: [23, 24, 26], [this chapter]

3.3 Problem Statement

In this section, we formally present the online scheduling model and the objective function. We also show a simple lower bound on the optimal makespan, against which the performance of our online algorithms will be measured.

3.3.1 Model and Objective

We consider the online scheduling of a ***Directed Acyclic Graph (DAG)*** of moldable tasks on a platform with P identical processors. Let $G = (V, E)$ denote the task graph, where $V = \{1, 2, \ldots, n\}$ represents a set of n tasks and $E \subseteq V \times V$ represents a set of precedence constraints (or dependencies) among the tasks. An edge $(i, j) \in E$ indicates that task j depends on task i, and therefore it cannot be executed before task i is completed. Task i is called the ***predecessor*** of task j, and task j is called the ***successor*** of task i. In this work, we do not consider the costs associated with the data transfers between dependent tasks.

The tasks are assumed to be ***moldable***, meaning that the number of processors allocated to a task can be determined by the scheduling algorithm at launch time, but once the task has started executing, its processor allocation cannot be changed. The execution time $t_j(p_j)$ of a task j is a function of the number p_j of processors allocated to it, and we assume that the processor allocation must be an integer between 1 and P. In this chapter, we focus on the first four speedup models of the previous papers, i.e.:

- *Roofline model*: linear speedup up to a bounded degree of parallelism $\bar{p}_j \in [1, P]$, i.e., $t_j(p_j) = w_j/p_j$ for $p_j \leq \bar{p}_j$, and $t_j(p_j) = w_j/\bar{p}_j$ for $p_j > \bar{p}_j$;
- *Communication model*: there is a communication overhead $c_j \geq 0$ per processor when more than one processor is used, i.e., $t_j(p_j) = w_j/p_j + (p_j - 1)c_j$;
- *Amdahl's model*: this is a particular case of the monotonic model with $t_j(p_j) = w_j(\frac{1-\gamma_j}{p_j} + \gamma_j)$, where $\gamma_j \in [0, 1]$ denotes the inherently sequential fraction of the job;
- *Mix model*: this mixed model combines Roofline, Communication and Amdahl's models with

$$t_j(p_j) = \frac{w_j(1-\gamma_j)}{\min(p, \bar{p}_j)} + w_j\gamma_j + (p_j - 1)c_j, \tag{3.1}$$

  which could capture more realistically the speedups of some complex applications.

From the execution time function of the task j, we can further define the ***area*** of the task as a function of the processor allocation as follows: $a_j(p_j) = p_j t_j(p_j)$. Intuitively, the

area represents the total amount of processor resources utilized over the entire period of task execution.

In this work, we consider the ***online scheduling*** model, where a task becomes available only when all of its predecessors have been completed. This represents a common scheduling model for ***dynamic task graphs***, whose dependencies are only revealed upon task completions [1, 34, 60, 97]. Furthermore, when a task j is available, all of its execution time parameters (i.e., w_j, $\bar{p}_j$, d_j, c_j) become known to the scheduling algorithm as well. The goal is to find a feasible schedule of the task graph that minimizes its overall completion time or ***makespan***, denoted by T. The performance of an online scheduling algorithm is measured by its competitive ratio: the algorithm is said to be ***c-competitive*** if, for any task graph, its makespan T is at most c times the makespan T^{OPT} produced by an optimal offline scheduler, i.e., $\frac{T}{T^{\text{OPT}}} \leq c$. Note that the optimal offline scheduler knows in advance all the tasks and their speedup models, as well as all dependencies in the graph. The competitive ratio is established against all possible strategies by an adversary trying to force the online algorithm to take bad decisions.

3.3.2 Lower Bound on Optimal Makespan

Given the execution time function in Equation (3.1), let us define $s_j = \sqrt{\frac{w_j}{c_j}}$. We can then compute the maximum number of processors that should be allocated to the task as:

$$p_j^{\max} = \min\left(P, \bar{p}_j, \tilde{p}_j\right) \ , \tag{3.2}$$

where

$$\tilde{p}_j = \begin{cases} \lfloor s_j \rfloor, & \text{if } t_j(\lfloor s_j \rfloor) \leq t_j(\lceil s_j \rceil) \\ \lceil s_j \rceil, & \text{otherwise} \end{cases}$$

Indeed, allocating more than $p_j^{\max}$ processors to the task will no longer decrease its execution time while only increasing its area. Thus, we can safely assume that the processor allocation of the task should never exceed $p_j^{\max}$ by any reasonable algorithm.

Furthermore, the task is said to satisfy the ***monotonic*** property [110] if the following two conditions hold:

- The execution time is a ***non-increasing*** function of the processor allocation, i.e., $t_j(p) \geq t_j(q)$ for all $1 \leq p < q \leq p_j^{\max}$;
- The area is a ***non-decreasing*** function of the processor allocation, i.e., $a_j(p) \leq a_j(q)$ for all $1 \leq p < q \leq p_j^{\max}$.

Note that the second condition above also suggests that the task cannot achieve superlinear speedup, i.e.,

$$\frac{t_j(p)}{t_j(q)} \leq \frac{q}{p} \text{ for all } 1 \leq p < q \leq p_j^{\max} \ . \tag{3.3}$$

Lemma 13. *A task j with execution time function given in Equation (3.1) is monotonic if its processor allocation is in the range $[1, p_j^{\max}]$.*

Proof. When the processor allocation is in the range $[1, p_j^{\max}]$, we have $p_j \leq p_j^{\max} \leq \bar{p}_j$. Thus, the execution time function simplifies to $t_j(p_j) = \frac{w_j}{p_j} + d_j + c_j(p_j - 1)$. This is a convex function whose minimum value is achieved at $\tilde{p}_j$. Since we also have $p_j \leq p_j^{\max} \leq \tilde{p}_j$, it shows that the execution time is a non-increasing function of p_j in the range $[1, p_j^{\max}]$.

Similarly, when $p_j \leq p_j^{\max} \leq \bar{p}_j$, the area becomes $a_j(p_j) = p_j t_j(p_j) = w_j + d_j p_j + c_j(p_j^2 - p_j)$, which is non-decreasing for any $p_j \geq 1$. □

Based on Lemma 13, the minimum execution time of the task is achieved as $t_j^{\min} = t_j(p_j^{\max})$ and the minimum area of the task is achieved as $a_j^{\min} = a_j(1)$. Further, we let t_j^{OPT} and a_j^{OPT} denote the execution time and area of the task under the processor allocation of an optimal schedule. We now define two quantities that can be used as a lower bound of the optimal makespan.

Definition 1. *Given the processor allocations of all the tasks in an optimal schedule,*

- *the* **total area** A^{OPT} *of the task graph is the sum of the areas of all the tasks in the graph, i.e.,* $A^{OPT} = \sum_{j=1}^{n} a_j^{OPT}$.
- *the length* $L^{OPT}(f)$ *of a path*[1] f *in the graph is the sum of the execution times of all the tasks along that path, i.e.,* $L^{OPT}(f) = \sum_{j \in f} t_j^{OPT}$. *The* **critical path length** C^{OPT} *of the graph is the longest length of any path in the graph, i.e.,* $C^{OPT} = \max_f L^{OPT}(f)$.

Clearly, the optimal makespan cannot be smaller than $\frac{A^{\text{OPT}}}{P}$ and C^{OPT}. This follows from the well-known area and critical-path bounds for scheduling any task graph [71]. The following lemma states this result.

Lemma 14. $T^{OPT} \geq \max\left(\frac{A^{OPT}}{P}, C^{OPT}\right)$.

3.4 A New Online Algorithm

In this section, we present a new online scheduling algorithm and derive its competitive ratio for the mix speedup model and its three special cases.

3.4.1 Algorithm Description

Algorithm 3 presents the pseudocode of the online scheduling algorithm, which at any time maintains the set of available tasks in a waiting queue Q. At time 0 or whenever a running task completes execution, it checks if new tasks have become available. If so, for each newly available task j, it finds a processor allocation p_j for the task (using Algorithm 4) before inserting it into the queue Q. Then, it applies the well-known ***list scheduling*** strategy [71] by scanning through all the available tasks in Q and executing each one right away if there are enough processors. Note that tasks are inserted into the queue without any priority considerations, although in practice certain priority rules may work better.

Algorithm 4 presents the details of the processor allocation strategy for any task j. It consists of two steps. The first step performs an initial allocation for the task, which is inspired by the ***local processor allocation*** strategy proposed in [23, 24]. Specifically, for each possible allocation $p \in [1, p_j^{\max}]$, we define the following:

- $g_j(p) \triangleq \frac{a_j(p)}{a_j^{\min}}$: ratio between the area of the task and its minimum area;
- $f_j(p) \triangleq \frac{t_j(p)}{t_j^{\min}}$: ratio between the execution time of the task and its minimum execution time.

We then find an allocation p that minimizes $f_j(p)$ subject to the constraint $g_j(p) \leq \alpha^M$, where $\alpha^M \geq 1$ is a constant whose exact value will be determined based upon the specific speedup model M under consideration. Since $g_j(p)$ is non-decreasing with p and $f_j(p)$ is non-increasing with p, the above optimization problem can be efficiently solved using binary search in $O(\log P)$ time.

[1] A path f consists of a sequence of tasks with linear dependency, i.e., $f = (j_{\pi(1)}, j_{\pi(2)}, \ldots, j_{\pi(v)})$, where the first task $j_{\pi(1)}$ in the sequence has no predecessor in the graph, the last task $j_{\pi(v)}$ has no successor, and, for each $2 \leq i \leq v$, task $j_{\pi(i)}$ is a successor of task $j_{\pi(i-1)}$.

Algorithm 3: Online_Scheduling_Algorithm

```
initialize a waiting queue Q
when at time 0 or a running task completes execution do
    // Processor Allocation
    for each new task j that becomes available do
        Allocate_Processor(j)
        insert task j into the waiting queue Q
    // List Scheduling
    for each task j in the waiting queue Q do
        if there are enough processors to execute the task then
            execute task j now
```

Algorithm 4: Allocate_Processor(j)

Input: Task j and its speedup model M; parameters α^M, μ^M
Output: Processor allocation p'_j for the task

// Step 1: Initial Allocation

Compute $p_j^{\max}$ based on Equation (3.2)
Compute $t_j^{\min} = t_j(p_j^{\max})$ and $a_j^{\min} = a_j(1)$
Find an allocation $p_j \in [1, p_j^{\max}]$ from the following optimization problem:

$$\min_p \ f_j(p) \triangleq \frac{t_j(p)}{t_j^{\min}}$$
$$\text{s.t. } g_j(p) \triangleq \frac{a_j(p)}{a_j^{\min}} \leq \alpha^M$$

// Step 2: Allocation Adjustment

if $p_j > \lceil \mu^M P \rceil$ **then** $p'_j \leftarrow \lceil \mu^M P \rceil$ **else** $p'_j \leftarrow p_j$

In the second step, the algorithm reduces the initial allocation to $\lceil \mu^M P \rceil$ if it is more than $\lceil \mu^M P \rceil$; otherwise the allocation will be unchanged. Here, $\mu^M \leq 0.5$ is also a constant whose value will be determined by the speedup model M. Let p_j denote the initial allocation for the task and p'_j the final allocation. Thus, after the second step, we have:

$$p'_j = \begin{cases} \lceil \mu^M P \rceil, & \text{if } p_j > \lceil \mu^M P \rceil \\ p_j, & \text{otherwise} \end{cases} . \tag{3.4}$$

This step adopts the technique first proposed in [110] and subsequently used in [93, 94]. The purpose is to enable the execution of more tasks at any time, thus potentially increasing the overall resource utilization of the platform and reducing the makespan.

The exact values of the two parameters α^M and μ^M for each speedup model M will be presented in Section 3.4.3 when analyzing the above algorithm.

3.4.2 A Novel Analysis Framework

We first outline an analysis framework, under which the competitive ratio of the proposed online algorithm will be derived for different speedup models. The framework is inspired by the analysis shown in [93, 94, 110] for list scheduling as well as the analysis used in [23, 24] for local processor allocation. Together, the result nicely connects the makespan of the online algorithm to the lower bound (Lemma 14), thus proving the competitive ratio.

Since the analysis framework in this section applies to any speedup model M, for simplicity, we will drop the superscript M for α^M and μ^M, and re-introduce it later in

Section 3.4.3 for each specific speedup model we consider.

Recall that T denotes the makespan of the online scheduling algorithm. Since the algorithm allocates and de-allocates processors upon task completions, the schedule can be divided into a set $\mathcal{I} = \{I_1, I_2, \dots\}$ of non-overlapping intervals, where tasks only start (or complete) at the beginning (or end) of an interval, and the number of utilized processors does not change during an interval. For each interval $I \in \mathcal{I}$, let $p(I)$ denote its processor utilization, i.e., the total number of processors used by all tasks running in interval I. We first classify the set of all intervals into the following two categories:

- $\mathcal{I}_0$: subset of intervals that satisfy $p(I) \in (0, \lceil (1-\mu)P \rceil)$;
- $\mathcal{I}_3$: subset of intervals that satisfy $p(I) \in [\lceil (1-\mu)P \rceil, P]$.

The following lemma shows a property for the subset of intervals in $\mathcal{I}_0$.

Lemma 15. *There exists a path f in the graph in which a task is always running in $\mathcal{I}_0$.*

Proof. During $\mathcal{I}_0$, the processor utilization is at most $\lceil (1-\mu)P \rceil - 1$, so there are at least $P - (\lceil (1-\mu)P \rceil - 1) \geq \lceil \mu P \rceil$ available processors. Based on Algorithm 4, any task is allocated at most $\lceil \mu P \rceil$ processors. Thus, there are enough processors to execute any new task (if one is available). This implies that there is no available task in the queue $\mathcal{Q}$ during $\mathcal{I}_0$. When a task graph is scheduled by the list scheduling algorithm, it is well known that there exists a path f in the graph such that some task along that path will be running whenever there is no available task in the queue [60, 94, 110], hence the result. □

Using the path f stated in Lemma 15, we further split $\mathcal{I}_0$ into the following two sub-categories:

- $\mathcal{I}_1$: subset of $\mathcal{I}_0$ where the processor allocation for the currently running task in f was not reduced (by the second step of Algorithm 4);
- $\mathcal{I}_2$: subset of $\mathcal{I}_0$ where the processor allocation for the currently running task in f was reduced (i.e., the task is running on $\lceil \mu P \rceil$ processors).

Finally, given the processor allocation of an optimal schedule, we further split $\mathcal{I}_2$ into the following two sub-categories:

- $\mathcal{I}_{2'}$: subset of $\mathcal{I}_2$ where the currently running task in f was allocated with strictly fewer processors than in the optimal schedule;
- $\mathcal{I}_{2''}$: subset of $\mathcal{I}_2$ where the currently running task in f was allocated with equal or more processors than in the optimal schedule.

Let $|I|$ denote the duration of an interval I, and let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$, $T_2 = \sum_{I \in \mathcal{I}_2} |I|$, $T_{2'} = \sum_{I \in \mathcal{I}_{2'}} |I|$, $T_{2''} = \sum_{I \in \mathcal{I}_{2''}} |I|$ and $T_3 = \sum_{I \in \mathcal{I}_3} |I|$ denote the total durations of the different categories of intervals, respectively. Since $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$ are obviously disjoint and partition $\mathcal{I}$, we have $T = T_1 + T_2 + T_3$. Finally, we define $z \in [0, 1]$ such that $T_{2'} = zT_2$ and $T_{2''} = (1-z)T_2$.

The next two lemmas relate these durations to the total area A_{OPT} and critical path length C_{OPT} of the task graph under an optimal schedule, given certain conditions on the initial processor allocations of the tasks under our algorithm.

Lemma 16. *If there exists a constant α such that, for each task j, its initial processor allocation satisfies $a_j(p_j) \leq \alpha a_j^{\min}$, then we have:*

$$\mu \left(z + \frac{1-z}{\alpha} \right) T_2 + \frac{(1-\mu)}{\alpha} T_3 \leq T^{\text{OPT}} . \tag{3.5}$$

Proof. As the area of each task j is non-decreasing with its processor allocation and $p'_j \leq p_j$, the final area of the task should satisfy $a_j(p'_j) \leq a_j(p_j) \leq \alpha a_j^{\min} \leq \alpha a_j^{\text{OPT}}$. Furthermore,

during $\mathcal{I}_{2'}$, any running task j from path f satisfies $a_j(p'_j) \leq a_j(p_j^{\text{OPT}}) = a_j^{\text{OPT}}$. We let $A_{2'|f}$ (resp. $A_{2''|f}$) denote the total area of the fraction of tasks from f running in $\mathcal{I}_{2'}$ (resp. $\mathcal{I}_{2''}$), and $A_{2'|f}^{\text{OPT}}$ (resp. $A_{2''|f}^{\text{OPT}}$) the corresponding fraction of area in an optimal schedule. We have $A_{2'|f} \leq A_{2'|f}^{\text{OPT}}$ and $A_{2''|f} \leq \alpha A_{2''|f}^{\text{OPT}}$. Since $\lceil \mu P \rceil \geq \mu P$ processors are used to run tasks from f in $\mathcal{I}_{2'} \cup \mathcal{I}_{2''}$, we have $\mu T_{2'} \leq \frac{A_{2'|f}}{P} \leq \frac{A_{2'|f}^{\text{OPT}}}{P}$ and $\mu T_{2''} \leq \frac{A_{2''|f}}{P} \leq \frac{\alpha A_{2''|f}^{\text{OPT}}}{P}$.

Finally, let A_3 denote the total area of the fraction of tasks running in $\mathcal{I}_3$ and A_3^{OPT} the corresponding fraction of area in an optimal schedule. Since at least $\lceil (1-\mu)P \rceil \geq (1-\mu)P$ processors are utilized during $\mathcal{I}_3$, we have $(1-\mu)T_3 \leq \frac{A_3}{P} \leq \frac{\alpha A_3^{\text{OPT}}}{P}$.

Thus, altogether we can derive:

$$\begin{aligned}
&\mu\left(z + \frac{1-z}{\alpha}\right) T_2 + \frac{(1-\mu)}{\alpha} T_3 \\
&= \mu T_{2'} + \frac{\mu T_{2''}}{\alpha} + \frac{(1-\mu)}{\alpha} T_3 \\
&\leq \frac{A_{2'|f}^{\text{OPT}}}{P} + \frac{A_{2''|f}^{\text{OPT}}}{P} + \frac{A_3^{\text{OPT}}}{P} \\
&\leq \frac{A^{\text{OPT}}}{P} \leq T^{\text{OPT}} .
\end{aligned}$$

The last inequality is due to the makespan lower bound shown in Lemma 14. □

Lemma 17. *If there exists a constant β such that, for each task j, its initial processor allocation satisfies $t_j(p_j) \leq \beta t_j^{\min}$, then we have:*

$$\frac{T_1}{\beta} + (\mu z + 1 - z) T_2 \leq T^{OPT} . \tag{3.6}$$

Proof. For any task j from path f running during $\mathcal{I}_1$, its processor allocation was not reduced by the second step of Algorithm 4, thus we must have $p'_j = p_j \leq \lceil \mu P \rceil$. Therefore, its execution time should satisfy $t_j(p'_j) = t_j(p_j) \leq \beta t_j^{\min} \leq \beta t_j^{\text{OPT}}$.

For any task j from path f running during $\mathcal{I}_{2'}$, its processor allocation has been reduced, i.e., $p'_j = \lceil \mu P \rceil$. Based on Equation (3.3), the task's execution time should satisfy $\frac{t_j(p'_j)}{t_j^{\min}} = \frac{t_j(\lceil \mu P \rceil)}{t_j(p_j^{\max})} \leq \frac{p_j^{\max}}{\lceil \mu P \rceil} \leq \frac{P}{\mu P} = \frac{1}{\mu}$. Thus, we have $t_j(p'_j) \leq \frac{1}{\mu} t_j^{\min} \leq \frac{1}{\mu} t_j^{\text{OPT}}$.

Finally, for any task j from path f running during $\mathcal{I}_{2''}$, its processor allocation is higher than that of an optimal schedule. Therefore, its execution time should satisfy: $t_j(p'_j) \leq t_j(p_j^{\text{OPT}}) = t_j^{\text{OPT}}$.

Now, let $L_{1|f}^{\text{OPT}}$ (resp. $L_{2'|f}^{\text{OPT}}$ and $L_{2''|f}^{\text{OPT}}$) denote the length for the portion of path f executed during $\mathcal{I}_1$ (resp. $\mathcal{I}_{2'}$ and $\mathcal{I}_{2''}$) under an optimal schedule. The argument above implies that $T_1 \leq \beta L_{1|f}^{\text{OPT}}$, $T_{2'} \leq \frac{1}{\mu} L_{2'|f}^{\text{OPT}}$ and $T_{2''} \leq L_{2''|f}^{\text{OPT}}$. Thus, we can derive:

$$\begin{aligned}
&\frac{T_1}{\beta} + (\mu z + 1 - z) T_2 \\
&= \frac{T_1}{\beta} + \mu T_{2'} + T_{2''} \\
&\leq L_{1|f}^{\text{OPT}} + L_{2'|f}^{\text{OPT}} + L_{2''|f}^{\text{OPT}} \\
&\leq L^{\text{OPT}}(f) \leq C^{\text{OPT}} \leq T^{\text{OPT}} .
\end{aligned}$$

The last inequality is again due to the makespan lower bound shown in Lemma 14. □

Based on the results of Lemmas 16 and 17, we can now derive an upper bound on the makespan of the online scheduling algorithm as shown below.

Lemma 18. *If there exist two constants α and β such that, for each task j, its initial processor allocation satisfies:*

$$g_j(p_j) \triangleq \frac{a_j(p_j)}{a_j^{\min}} \leq \alpha \ , \tag{3.7}$$

$$f_j(p_j) \triangleq \frac{t_j(p_j)}{t_j^{\min}} \leq \beta \ , \tag{3.8}$$

then by setting μ such that $\beta + \frac{\alpha}{1-\mu} = \frac{1}{\mu}$, i.e., $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta}$, and under the condition $\beta \geq \frac{\mu(\alpha-1)}{(1-\mu)^2}$, we get:

$$\frac{T}{T^{\text{OPT}}} \leq \frac{1}{\mu} = \frac{2\beta}{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}} \ . \tag{3.9}$$

Proof. As the makespan is given by $T = T_1 + T_2 + T_3$, we can multiply both sides by $\frac{1-\mu}{\alpha}$ and apply Equation (3.5) to remove the T_3 term, which gives:

$$\frac{1-\mu}{\alpha}T \leq \frac{1-\mu}{\alpha}T_1 + \left(\frac{1-\mu-z\alpha\mu-(1-z)\mu}{\alpha}\right)T_2 + T^{\text{OPT}} \ .$$

We can then multiply both sides of the above inequality by $\frac{\alpha}{(1-\mu)\beta}$ and apply Equation (3.6) to remove the T_1 term. This gives:

$$\frac{T}{\beta} \leq \left(\frac{1-\mu-z\alpha\mu-(1-z)\mu}{(1-\mu)\beta} - \mu z + z - 1\right)T_2 + \left(1 + \frac{\alpha}{(1-\mu)\beta}\right)T^{\text{OPT}} \ .$$

Now, if $f(z) = \frac{1-\mu-z\alpha\mu-(1-z)\mu}{(1-\mu)\beta} - \mu z + z - 1 \leq 0$ is true for all $z \in [0,1]$, we can omit the T_2 term in the above inequality and get:

$$T \leq \left(\beta + \frac{\alpha}{(1-\mu)}\right)T^{\text{OPT}} = \frac{1}{\mu}T^{\text{OPT}} \ .$$

We have $f'(z) = \frac{\mu(1-\alpha)}{(1-\mu)\beta} + (1-\mu) \geq 0$ under the condition $\beta \geq \frac{\mu(\alpha-1)}{(1-\mu)^2}$, which makes $f(z)$ an increasing function of z. Thus, we simply need to ensure that $f(1) = \frac{1-\mu-\alpha\mu}{(1-\mu)\beta} - \mu \leq 0$, which is true if $\beta + \frac{\alpha}{1-\mu} = \frac{1}{\mu}$. One can then solve for μ from the above second-degree equation, and get $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta}$.

Finally, we show that the value of μ above is well-defined and is a valid choice satisfying $\mu \in (0, 0.5]$.

- First, we can derive that $\Delta = (\alpha+\beta+1)^2 - 4\beta > (\beta+1)^2 - 4\beta = (\beta-1)^2 \geq 0$. Thus, the value of μ is well-defined.
- We have $\mu > \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2}}{2\beta} = 0$, since $\alpha, \beta > 0$.
- We can show $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta} \leq 0.5$, which after some manipulations is equivalent to showing $0 \leq \beta^2 + 2\beta(\alpha-1)$. The latter inequality is always true since $\alpha \geq 1$. □

Remarks. We point out that the two bounds shown in Lemma 18, i.e., Inequalities (3.7) and (3.8), must be satisfied by all tasks in a task graph for the derived competitive ratio shown in Equation (3.9) to hold. In Section 3.4.3, we will prove that, for a given speedup model and regardless of the task parameters, there always exists a processor allocation that satisfies the two bounds with a particular (α, β) choice that achieves the minimum (or close to minimum) competitive ratio. We then show in Section 3.5 that the obtained competitive ratios are tight (or almost tight) by proving matching (or nearly matching) lower bounds for different speedup models. Thus, given an (α, β) pair for a speedup model, the processor allocation algorithm should find an allocation for each task to satisfy the two bounds. However, there could be multiple allocations that all satisfy the bounds. Among these allocations, Algorithm 4 finds one that, subject to the α bound, minimizes the execution time of the task, thus satisfying the β bound as well. Intuitively, this is a good practical choice and it also helps to simplify the analysis, which we will present in Section 3.4.3.

3.4.3 Competitive Ratios

In this section, we prove the competitive ratios for the online algorithm under different speedup models. Based on Lemma 18, the competitive ratio is given by: $\frac{1}{\mu} = \frac{2\beta}{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}$ subject to the constraint $\beta \geq \frac{\alpha-1}{(1-\mu)^2}$.

For a given speedup model with parameters $\mathcal{P} \subseteq \{w, d, c, \bar{p}\}$, a generic approach for minimizing the ratio above can be outlined as follows: First, compute $\beta(\alpha)$ as small as possible based on the model parameters $\mathcal{P}$ for any fixed $\alpha \geq 1$. To do that, since the area of a task is non-decreasing with the processor allocation and the time non-increasing in the range $[1, p^{\max}]$, we can find the largest processor allocation $p^*(\mathcal{P}) \in [1, p^{\max}]$ that satisfies $\frac{a(p^*(\mathcal{P}))}{a^{\min}} \leq \alpha$ and then compute $\beta(\alpha) = sup_{\mathcal{P}} \left(\frac{t(p^*(\mathcal{P}))}{t^{\min}}\right)$. Finally, plug $\beta(\alpha)$ into the expression of the competitive ratio and find the α that minimizes it while satisfying the constraint.

Although the technique outlined above is a good generic approach, the computations involved are often too complicated for some speedup models and solving it will rely on numerical tools. Therefore, to derive the competitive ratios analytically, we will simply find a valid pair (α, β) below for each considered speedup model while verifying that the constraint is satisfied. We will then show the tightness (or near tightness) of the obtained competitive ratios by computing the lower bounds in Section 3.5.

As in previous paper, we now re-introduce superscript $M \in \{\text{Roo}, \text{Com}, \text{Amd}, \text{Mix}\}$ to the notations α^M, β^M and μ^M in the lemmas and theorems below. Given a speedup model M, the analysis focuses on finding α^M and β^M for each individual task, thus we will drop the task index j for simplicity.

Roofline Model

Recall that a task follows the roofline speedup model if its execution time satisfies $t(p) = \frac{w}{\min(p,\bar{p})}$ for some $\bar{p} \leq P$.

Lemma 19. *For any task that follows the roofline speedup model, there exists a processor allocation that achieves $\alpha^{\text{Roo}} = \beta^{\text{Roo}} = 1$.*

Proof. For any task with $\bar{p}$, setting the processor allocation to $p = \bar{p}$ clearly achieves the minimum execution time $t^{\min} = \frac{w}{\bar{p}}$ for the task. It also achieves the minimum area

$a^{\min} = w$, which is not affected by the processor allocation in $[1, \bar{p}]$ due to the task's linear speedup in this range. Thus, this gives $\alpha^{\text{Roo}} = \beta^{\text{Roo}} = 1$. □

Theorem 12. *Algorithm 3 is* $\frac{2}{3-\sqrt{5}} < 2.62$*-competitive for any graph of tasks that follow the roofline speedup model. This is achieved with* $\mu^{\text{Roo}} = \frac{3-\sqrt{5}}{2} \approx 0.382$.

Proof. With $\alpha = \beta = 1$, the constraint $\beta \geq \frac{\mu(\alpha-1)}{(1-\mu)^2} = 0$ is obviously satisfied. Thus, we get $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta} = \frac{3-\sqrt{5}}{2}$, and the competitive ratio is given by $\frac{1}{\mu} = \frac{2}{3-\sqrt{5}} < 2.62$. □

Remarks. The above ratio retains the same result by Feldmann et al. [60][2], who also proved a matching lower bound for any online deterministic algorithm under the "non-clairvoyant" setting, where the work w of a task is unknown to the scheduler. In Section 3.5, we will prove the same lower bound, but without the non-clairvoyant setting for a class of list scheduling algorithms with deterministic local decisions for processor allocation.

Communication Model

Remarks. Using the generic approach outlined at the beginning of this section, we could actually compute the best possible $\beta(\alpha)$ for any $\alpha > 1$.

The analysis, however, is very technical, and finding the optimal α then requires numerical analysis tools. It turns out that the best (α, β) pair is $(\frac{4}{3}, \frac{3}{2})$, and we will show that this pair is optimal in Section 3.5.

Theorem 13. *Algorithm 3 is* $\frac{18}{23-\sqrt{313}} < 3.391$*-competitive for any graph of tasks that follow the communication model. This is achieved with* $\mu^{\text{Com}} = \frac{23-\sqrt{313}}{18} \approx 0.295$.

Proof. We use Lemma 4 giving $\alpha = \frac{4}{3}$ and $\beta = \frac{3}{2}$, pluf it to $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta} = \frac{23-\sqrt{313}}{18}$, and the constraint $\beta \geq \frac{\mu(\alpha-1)}{(1-\mu)^2} \approx 0.2$ is satisfied. Thus, the competitive ratio is given by $\frac{1}{\mu} = \frac{18}{23-\sqrt{313}} < 3.391$. □

Amdahl's Model

Recall that a task follows the Amdahl's model if its execution time function is $t(p) = \frac{w}{p} + d$, with $d \geq 0$, thus the area function is given by $a(p) = pt(p) = w + dp$.

Lemma 20. *For any task that follows the Amdahl's model, there exists a processor allocation that achieves* $\alpha^{\text{Amd}} = \frac{\sqrt{2}+1+\sqrt{2\sqrt{2}-1}}{2} \approx 1.883$ *and* $\beta^{\text{Amd}} = \frac{1+\sqrt{4\sqrt{2}+5}}{2} \approx 2.132$.

Proof. In Lemma 5, we have shown that for any $\alpha > 1$, we could find a processor allocation such that $\beta \leq \frac{\alpha}{\alpha-1} \triangleq \beta(\alpha)$. We can now substitute $\beta(\alpha)$ into the expression of the competitive ratio and get:

$$\frac{1}{\mu} = \frac{2\frac{\alpha}{\alpha-1}}{\alpha + \frac{\alpha}{\alpha-1} + 1 - \sqrt{\left(\alpha + \frac{\alpha}{\alpha-1} + 1\right)^2 - 4\frac{\alpha}{\alpha-1}}}.$$

[2] In [60], each task has a parallelism p, and can be virtualized if $p' \leq p$ processors are used for execution, with a linear slowdown. This is equivalent to the roofline model.

To minimize the ratio above, one can use the standard technique of differentiating and setting the derivative to zero. The expression is quite long, and the full analysis is omitted. It turns out that $\alpha^{\text{AMD}} = \frac{\sqrt{2}+1+\sqrt{2\sqrt{2}-1}}{2}$ minimizes the ratio (again, Section 3.5 will prove the optimality of this choice anyway). Plugging it back into $\beta(\alpha) = \frac{\alpha}{\alpha-1}$ and simplifying, we can get $\beta^{\text{AMD}} = \frac{1+\sqrt{4\sqrt{2}+5}}{2}$. □

Theorem 14. *Algorithm 3 is* $\frac{2}{1-\sqrt{8\sqrt{2}-11}} < 4.55$*-competitive for any graph of tasks that follow the Amdahl's model. This is achieved with* $\mu^{\text{AMD}} = \frac{1-\sqrt{8\sqrt{2}-11}}{2} \approx 0.22$.

Proof. By substituting $\alpha = \frac{\sqrt{2}+1+\sqrt{2\sqrt{2}-1}}{2}$ and $\beta = \frac{1+\sqrt{4\sqrt{2}+5}}{2}$ into the expression of μ and simpifying (hard!), we can get $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta} = \frac{1-\sqrt{8\sqrt{2}-11}}{2}$. We can also check that the constraint $\beta > \frac{\mu(\alpha-1)}{(1-\mu)^2} \approx 0.32$ is satisfied. Thus, the competitive ratio is given by $\frac{1}{\mu} = \frac{2}{1-\sqrt{8\sqrt{2}-11}} < 4.55$. □

Remark 2. *Another way to find the expression of* μ^{AMD} *is to show that* α^{AMD} *minimizes the ratio* $\beta^{\text{AMD}} + \frac{\alpha^{\text{AMD}}}{1-\mu^{AMD}} = \frac{\alpha^{\text{AMD}}}{\alpha^{\text{AMD}}-1} + \frac{\alpha^{\text{AMD}}}{1-\mu^{\text{AMD}}}$*. By deriving and simplifying, we can show the much easier expression* $\mu^{\text{AMD}} = (\alpha^{\text{AMD}} - 1)^2$ *in the Amdahl's case.*

Mix Model

We finally consider the mix speedup model given by $t_j(p_j) = \frac{w_j(1-d_j)}{\min(p,\bar{p}_j)} + w_j d_j + (p_j - 1)c_j$. Without loss of generality, we assume $w > 0$ (otherwise we get $\alpha = \beta = 1$ using one processor), and $c, d > 0$ (otherwise the model reduces to the communication or the Amdahl's model and also results in smaller α and β). We rewrite the execution time function as: $t(p) = c\left(\frac{w'}{\min(p,\bar{p})} + d' + p - 1\right)$ with $w' = \frac{w}{c}$ and $d' = \frac{d}{c}$. We further assume $\bar{p} \leq P$ (otherwise changing $\bar{p}$ to P does not affect the execution time of the task for any feasible processor allocation). Finally, any reasonable scheduling algorithm will not allocate more than $\bar{p}$ processors to the task since it would increase both execution time and area. Thus, assuming $p \leq \bar{p}$, we can simplify the execution time function as $t(p) = c\left(\frac{w'}{p} + d' + p - 1\right)$ and the area function is given by $a(p) = c(w' + d'p + p(p-1))$.

Theorem 15. *Algorithm 3 is* $\frac{27}{33-\sqrt{738}} < 4.63$*-competitive for any graph of tasks that follow the mix speedup model given in Equation (3.1). This is achieved with* $\mu^{\text{MIX}} = \frac{33-\sqrt{738}}{27} \approx 0.216$.

Proof. With $\alpha = 2$ and $\beta = \frac{27}{13}$, we get $\mu = \frac{\alpha+\beta+1-\sqrt{(\alpha+\beta+1)^2-4\beta}}{2\beta} = \frac{33-\sqrt{738}}{27}$ and the constraint $\beta \geq \frac{\mu(\alpha-1)}{(1-\mu)^2} \approx 0.35$ is satisfied. Thus, the competitive ratio is given by $\frac{1}{\mu} = \frac{27}{33-\sqrt{738}} < 4.63$. □

Finally, Table III summarizes the parameters and competitive ratios derived for all the considered speedup models.

Table III: Summary of parameters and competitive ratios for different speedup models.

Model M	μ^M	α^M	β^M	**Competitive Ratio**
Roofline (Roo)	$\frac{3-\sqrt{5}}{2}$	1	1	$\frac{2}{3-\sqrt{5}} \approx 2.62$
Com. (Com)	$\frac{23-\sqrt{313}}{18}$	$\frac{4}{3}$	$\frac{3}{2}$	$\frac{18}{23-\sqrt{313}} \approx 3.39$
Amdahl (Amd)	$\frac{1-\sqrt{8\sqrt{2}-11}}{2}$	$\frac{\sqrt{2}+1+\sqrt{2\sqrt{2}-1}}{2}$	$\frac{1+\sqrt{4\sqrt{2}+5}}{2}$	$\frac{2}{1-\sqrt{8\sqrt{2}-11}} \approx 4.55$
Mix (Mix)	$\frac{33-\sqrt{738}}{27}$	2	$\frac{27}{13}$	$\frac{27}{33-\sqrt{738}} \approx 4.63$

3.5 Lower Bounds for Online List Scheduling Algorithms with Deterministic Local Decisions

In Section 3.4.3, we derived the competitive ratios of our online algorithm under several common speedup models. In this section, we will show corresponding lower bounds on the competitive ratios.

Definition 2. *An online algorithm is said to make* ***deterministic local decisions*** *if it allocates processors by considering only P and the parameters of a task's speedup function (i.e., w, $\bar{p}$, d, c). Thus, two identical tasks will receive exactly the same allocation regardless of their relative positions in the task graph as well as the graph structure.*

Ultimately, we will show that our algorithm has the optimal competitive ratios over all algorithms in this class for the roofline, communication, and Amdahl's models. The result also indicates that our algorithms's competitive ratio for the mix model is close to optimal using the lower bound of the Amdahl's model.

3.5.1 Analysis Overview

Under any model M, we have shown in Section 3.4 that, for any task, our online algorithm achieves:

$$\frac{a}{a^{\min}} \leq \alpha^M \quad \text{and} \quad \frac{t}{t^{\min}} \leq \beta^M \ , \tag{3.10}$$

where $\frac{\alpha^M}{1-\mu^M} + \beta^M = \frac{1}{\mu^M}$. In particular, for any possible instance consisting of a set $\mathcal{T}$ of tasks, if our algorithm achieves a makespan of T and the optimal makespan is T^{Opt}, then we have shown that:

$$\frac{T}{T^{\text{Opt}}} \leq \max_{j \in \mathcal{T}} \frac{a_j}{a_j^{\min}(1-\mu^M)} + \max_{j \in \mathcal{T}} \frac{t_j}{t_j^{\min}} \leq \frac{\alpha^M}{1-\mu^M} + \beta^M = \frac{1}{\mu^M} \ . \tag{3.11}$$

Two-step Approach

To prove the lower bounds, we will proceed in two steps corresponding to proving the tightness of the above two inequalities (3.10) and (3.11), respectively. More precisely, we will fix a model M and suppose an online list scheduling algorithm $\mathcal{A}$ respecting Definition 2 and having a competitive ratio strictly less than $\frac{1}{\mu^M}$ exists. Specifically, we will assume

that $\mathcal{A}$'s competitive ratio is $\frac{1}{\mu^M} - 4\epsilon$ for some $0 < \epsilon < 1$. In the first step (Section 3.5.2), we will show the existence of two tasks A and B as well as another algorithm $\mathcal{A}^*$ such that:

$$\frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} \geq \frac{1}{\mu^M} - \epsilon \ , \tag{3.12}$$

thus showing the tightness of our local analysis. In the second step (Section 3.5.3), we will build an instance using these two tasks (as well as two other tasks C and D) such that, by using list scheduling, algorithm $\mathcal{A}$ has no choice but to achieve a makespan that satisfies:

$$\frac{T}{T^*} \geq \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 2\epsilon \geq \frac{1}{\mu^M} - 3\epsilon \ , \tag{3.13}$$

where T and T^* denote the makespans of $\mathcal{A}$ and $\mathcal{A}^*$ for this instance, respectively. This contradicts the assumed competitive ratio, thus showing the tightness of our global analysis and hence the non-existence of algorithm $\mathcal{A}$.

Notations

We will use four different tasks A, B, C, D to construct the lower bound instances. For algorithm $\mathcal{A}$, we let p_A (resp. p_B, p_C, p_D) denote its processor allocation for task A (resp. B, C, D), let t_A (resp. t_B, t_C, t_D) denote the resulting execution time of the tasks, and let $a_A = t_A p_A$ (resp. a_B, a_C, a_D) denote the resulting area of the tasks. Similarly, we use $p_A^*, p_B^*, p_C^*, p_D^*, t_A^*, t_B^*, t_C^*, t_D^*, a_A^*, a_B^*, a_C^*, a_D^*$ to denote the corresponding values for algorithm $\mathcal{A}^*$.

Constraints

The lower bound instances need to respect a set of constraints (or rules) for the tasks and for the task graph, which are required to show the global results. This will allow us to prove the lower bound regardless of the model, as long as these constraints are satisfied. For convenience, Table IV lists and labels all the required constraints (R's) along with some definitions (F's). We will refer to them according to their labels, and use $(R : \checkmark)$ to denote that a constraint R is satisfied in the subsequent analysis.

3.5.2 Step 1: Local Analysis

In this section, we will show that, for a given model M and any online list scheduling algorithm $\mathcal{A}$ respecting Definition 2, there exist tasks A and B as well as another algorithm $\mathcal{A}^*$ such that $\frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} \geq \frac{1}{\mu^M} - \epsilon$. We start with the following theorem and will prove it separately for each considered model.

Theorem 16. *Given a model $M \in \{\textsc{Roo}, \textsc{Com}, \textsc{Amd}\}$, let $\mathcal{A}$ be an online list scheduling algorithm respecting Definition 2 with a competitive ratio of $\frac{1}{\mu^M} - 4\epsilon$ for some $0 < \epsilon < 1$, and let $P \geq \left(\frac{120900}{\epsilon}\right)^4$. Then, there exist four tasks A, B, C and D satisfying the constraints on tasks in Table IV (R_1 to R_{12}) and another algorithm $\mathcal{A}^*$ such that:*

$$\frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} \geq \frac{1}{\mu^M} - \epsilon \ . \tag{3.14}$$

Proof. First, we set $t_C(p) = \frac{\epsilon}{121P^2 \cdot p}$. Indeed, this execution time function belongs to all speedup models[3], and clearly, we have $(R_{11} : \checkmark)$, i.e., constraint (R_{11}) is satisfied. Further,

[3]For all models, we have $w = \frac{\epsilon}{121P^2}$. Additionally, for the roofline model, $\bar{p} = \infty$; for the communication model, $c = 0$; and for the Amdhal's model, $d = 0$.

Table IV: List of constraints (R's) and definitions (F's) for constructing lower bound instances.

For tasks	**For task graph**
$p_A^* \le P^{3/4}$ (R_1)	$P \ge \left(\frac{120900}{\epsilon}\right)^4$ (F_1)
$0.1 \le t_B^* \le 100$ (R_2)	$X = \left\lceil \frac{P - p_C + 1}{p_B} \right\rceil$ (F_2)
$p_B \le P^{3/4}$ (R_3)	$K = \left\lceil \frac{5t_A^*}{\epsilon X t_B^*} \right\rceil$ (F_3)
$t_D \le t_B$ (R_4)	$Y = \left\lfloor \frac{XKt_B^*}{t_A^*} \right\rfloor$ (F_4)
$t_D^* \le \frac{\epsilon}{121P^2}$ (R_5)	$Z = K(P - p_A^*)$ (F_5)
$p_D \le 4$ (R_6)	
$t_A^* \le 24 t_B^*$ (R_7)	$1 \le X \le P$ (R_{13})
$t_B^* = a_B^* = t_B(1)$ (R_8)	$XKt_B^*\left(1 - \frac{\epsilon}{5}\right) \le Yt_A^* \le XKt_B^*$ (R_{14})
$t_A \le 5t_A^*$ (R_9)	$K(P - P^{3/4}) \le Z \le \frac{121P}{\epsilon}$ (R_{15})
$a_B \le 5a_B^*$ (R_{10})	
$t_C^* \le \frac{\epsilon}{121P^2}$ (R_{11})	
$p_C \ge \mu^M P$ (R_{12})	

if we had $p_C < \mu^M P$, then we would have $\frac{t_C}{t_C^*} > \frac{\frac{\epsilon}{121P^2 \cdot \mu^M P}}{\frac{\epsilon}{121P^2 \cdot P}} = \frac{1}{\mu^M}$, which contradicts the competitive ratio of $\mathcal{A}$ on an instance consisting of only one task C (R_{12} : ✓). Similarly, for task A, we must have $t_A \le 5t_A^*$ to respect the competitive ratio of $\mathcal{A}$ on an instance consisting of a single such task, as $\frac{1}{\mu^M} < 5$ for all models (R_9 : ✓). For task B, we will set $p_B^* = 1$ for all models (R_8✓). Also, if we had $a_B > 5a_B^*$, then an instance consisting of P independent such tasks would result in a makespan at least $\frac{Pa_B}{P} = a_B$ for $\mathcal{A}$, since Pa_B is the total area to be completed on P processors, while $\mathcal{A}^*$ can execute all tasks simultaneously in parallel with a resulting makespan of a_B^*, which also contradicts the competitive ratio of $\mathcal{A}$ (R_{10} : ✓).

The following three lemmas will conclude the proof of the theorem by considering each of the three models separately. Note that we only need to define tasks A, B and D, and verify the constraints (R_1) to (R_7). □

Lemma 21. *Theorem 16 is true for the roofline model.*

Proof. For the roofline model, we will only use sequential tasks with $\bar{p} = 1$, and set $t_A(p) = t_B(p) = 1$ and $t_D(p) = \frac{\epsilon}{121P^2}$ for all p. Thus, we have (R_2, R_4, R_5, R_7 : ✓). Clearly, if $\mathcal{A}$ allocates 3 or more processors to task B or D, then running P independent such tasks would result in a makespan at least 3 times that of the optimal using a single processor, constradicting the competitive ratio of $\mathcal{A}$. Thus, we can assume that $p_B \le 2 \le P^{3/4}$ (R_3 : ✓) and $p_D \le 2 \le 4$ (R_6 : ✓). We further set $p_A^* = p_B^* = 1$ (R_1 : ✓). These give $\frac{a_B}{a_B^*} \ge 1$ and $\frac{t_A}{t_A^*} = 1$. With $\mu^{\text{Roo}} = \frac{3+\sqrt{5}}{2}$, we obtain:

$$\frac{a_B}{a_B^*(1 - \mu^{\text{Roo}})} + \frac{t_A}{t_A^*} \ge \frac{1}{1 - \mu^{\text{Roo}}} + 1 = \frac{1}{\mu^{\text{Roo}}} \ .$$ □

Lemma 22. *Theorem 16 is true for the communication model.*

Proof. Given algorithm $\mathcal{A}$, we define $\mathcal{U}$ to be the subset of $\{x \in \mathbb{R}^+\}$ such that $\mathcal{A}$ allocates one processor to a task whose execution time function has the form: $t(p) = \frac{x}{p} + p - 1$. By definition, $\mathcal{A}$ always has the same allocation for identical tasks, so $\mathcal{U}$ is well-defined and must satisfy:

- $[0, 0.1] \subseteq \mathcal{U}$, otherwise there would exist an $x \leq 0.1$ such that $\mathcal{A}$ allocates at least two processors for $t(p) = \frac{x}{p} + p - 1$, and we would have $\frac{a}{a^{\min}} \geq \frac{a(2)}{a(1)} = \frac{x+2}{x} = 1 + \frac{2}{x} \geq 21$, which contradicts the competitive ratio of $\mathcal{A}$.
- $\mathcal{U} \subseteq [0, 64]$, otherwise there would exist an $x > 64$ such that $\mathcal{A}$ allocates one processor for $t(p) = \frac{x}{p} + p - 1$, and we would have $\frac{t}{t^{\min}} \geq \frac{x}{x/8+7} = \frac{8x+448}{x+56} - \frac{448}{x+56} \geq 8 - \frac{448}{120} > 4$, which also contradicts the competitive ratio of $\mathcal{A}$.

Based on the previous analysis, we now consider $s = sup(\mathcal{U}) \in [0.1, 64]$. By definition, there exists a $\delta \in [0, \frac{1}{P})$ such that $(s - \delta) \in \mathcal{U}$ and $(s - \delta + \frac{1}{P}) \notin \mathcal{U}$. We choose such δ, and set $\bar{w} = s - \delta > 0.09$, $t_A(p) = \frac{\bar{w}}{p} + p - 1$ and $t_B(p) = \frac{\bar{w}+\frac{1}{P}}{p} + p - 1$ with $p_A = 1$ and $p_B > 1$. Thus, we get $t_B^* = \bar{w} + \frac{1}{P}$ $(R_2 : \checkmark)$.

We also have $\frac{a_B}{a_B^{\min}} = \frac{\bar{w}+\frac{1}{P}+p_B(p_B-1)}{\bar{w}+\frac{1}{P}} \geq 1 + \frac{(p_B-1)^2}{65}$, from which we can assume $p_B \leq 15$, otherwise $\frac{a_B}{a_B^{\min}} > 4$, again contradicting $\mathcal{A}$'s competitive ratio. As $P > 81$, it leads to $p_B \leq P^{3/4}$ $(R_3 : \checkmark)$. We now show that Inequality (3.14) is true:

$$
\begin{aligned}
\frac{a_B}{a_B^*(1-\mu^{\text{COM}})} + \frac{t_A}{t_A^*} &\geq \frac{\bar{w}+2}{(\bar{w}+\frac{1}{P})(1-\mu^{\text{COM}})} + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&= \frac{\bar{w}+2}{\bar{w}(1-\mu^{\text{COM}})} \cdot \frac{1}{1+\frac{1}{P\bar{w}}} + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&\geq \frac{\bar{w}+2}{\bar{w}(1-\mu^{\text{COM}})} \cdot \left(1 - \frac{1}{P\bar{w}}\right) + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&= \frac{\bar{w}+2}{\bar{w}(1-\mu^{\text{COM}})} - \frac{(\bar{w}+2)}{P\bar{w}^2(1-\mu^{\text{COM}})} + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&\geq \frac{\bar{w}+2}{\bar{w}(1-\mu^{\text{COM}})} - \frac{2(\bar{w}+2)}{P\bar{w}^2} + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&\geq \frac{\bar{w}+2}{\bar{w}(1-\mu^{\text{COM}})} - \frac{16297}{P} + \frac{\bar{w}}{\frac{\bar{w}}{p_A^*} + p_A^* - 1} \\
&\geq \frac{1+\frac{2}{\bar{w}}}{1-\mu^{\text{COM}}} + \frac{1}{\frac{1}{p_A^*} + \frac{p_A^*-1}{\bar{w}}} - \epsilon \ .
\end{aligned}
$$

We further consider two cases:

Case 1: $\bar{w} \leq 6$. In this case, we set $p_A^* = 2$ and obtain:

$$
\frac{a_B}{a_B^*(1-\mu^{\text{COM}})} + \frac{t_A}{t_A^*} \geq \frac{1+\frac{2}{\bar{w}}}{1-\mu^{\text{COM}}} + \frac{1}{\frac{1}{2} + \frac{1}{\bar{w}}} - \epsilon \ .
$$

We define $f(\bar{w}) \triangleq \frac{1+\frac{2}{\bar{w}}}{1-\mu^{\text{COM}}} + \frac{1}{\frac{1}{2}+\frac{1}{\bar{w}}}$, and get $f'(\bar{w}) = \frac{4}{(\bar{w}+2)^2} - \frac{2}{(1-\mu^{\text{COM}})\bar{w}^2}$, which is negative in $[0, w^0)$, where w^0 is the smallest positive $\bar{w}$ such that $f'(\bar{w}) = 0$. Solving the equation

above, we find $w^0 = \frac{2+2\sqrt{2-2\mu^{\text{COM}}}}{1-2\mu^{\text{COM}}} > 6$. Therefore, we can replace $\bar{w}$ by 6 to obtain:

$$\begin{aligned}\frac{a_B}{a_B^*(1-\mu^{\text{COM}})} + \frac{t_A}{t_A^*} &\geq \frac{4}{3(1-\mu^{\text{COM}})} + \frac{3}{2} - \epsilon \\ &= \frac{\alpha^{\text{COM}}}{1-\mu^{\text{COM}}} + \beta^{\text{COM}} - \epsilon \\ &= \frac{1}{\mu^{\text{COM}}} - \epsilon \ .\end{aligned}$$

Case 2: $\bar{w} > 6$. In this case, we set $p_A^* = 3$ and obtain:

$$\frac{a_B}{a_B^*(1-\mu^{\text{COM}})} + \frac{t_A}{t_A^*} \geq \frac{1+\frac{2}{\bar{w}}}{1-\mu^{\text{COM}}} + \frac{1}{\frac{1}{3}+\frac{2}{\bar{w}}} - \epsilon \ .$$

We again define $f(\bar{w}) \triangleq \frac{1+\frac{2}{\bar{w}}}{1-\mu^{\text{COM}}} + \frac{1}{\frac{1}{3}+\frac{2}{\bar{w}}}$, and get $f'(\bar{w}) = \frac{18}{(\bar{w}+6)^2} - \frac{2}{(1-\mu^{\text{COM}})\bar{w}^2}$, which is positive in (w^0, ∞), where w^0 is the largest positive $\bar{w}$ such that $f'(\bar{w}) = 0$. Solving the equation above, we find $w^0 = \frac{6+18\sqrt{1-\mu^{\text{COM}}}}{8-9\mu^{\text{COM}}} < 6$. Therefore, we can replace $\bar{w}$ by 6 to obtain:

$$\begin{aligned}\frac{a_B}{a_B^*(1-\mu^{\text{COM}})} + \frac{t_A}{t_A^*} &\geq \frac{4}{3(1-\mu^{\text{COM}})} + \frac{3}{2} - \epsilon \\ &= \frac{\alpha^{\text{COM}}}{1-\mu^{\text{COM}}} + \beta^{\text{COM}} - \epsilon \\ &= \frac{1}{\mu^{\text{COM}}} - \epsilon \ .\end{aligned}$$

In both cases, we get the desired result with ($R_1 : \checkmark$). Further, because $p_A^* \leq 3$ and $\bar{w} > 0.09$, we get $t_A^* \leq \bar{w} + 2 \leq 24\bar{w} \leq 24 t_B^*$ ($R_7 : \checkmark$).

Finally, for task D, we set $t_D(p) = \frac{\epsilon}{121P^2 \cdot p} + p - 1$ and $p_D^* = 1$. Thus, $\mathcal{A}$ must also allocate one processor to the task (i.e., $P_D = 1$), otherwise $\frac{a_D}{a_D^{\min}} \geq \frac{\frac{\epsilon}{121P^2}+2}{\frac{\epsilon}{121P^2}} > 5$, which contradicts its competitive ratio. Therefore, we have ($R_4, R_5, R_6 : \checkmark$). □

Lemma 23. *Theorem 16 is true for the Amdahl's model.*

Proof. Given algorithm $\mathcal{A}$, we define $\mathcal{U}$ to be the subset of $\{x \in \mathbb{R}^+\}$ such that $\mathcal{A}$ allocates strictly less than $\sqrt{P}$ processors to a task whose execution time function has the form: $t(p) = \frac{x}{p} + \frac{1}{\sqrt{P}}$. By definition, $\mathcal{A}$ always has the same allocation for identical tasks, so $\mathcal{U}$ is well-defined and must satisfy:

- $[0, 0.1] \subseteq \mathcal{U}$, otherwise there would exist an $x \leq 0.1$ such that $\mathcal{A}$ allocates at least $\sqrt{P}$ processors for $t(p) = \frac{x}{p} + \frac{1}{\sqrt{P}}$, and we would have $\frac{a}{a^{\min}} \geq \frac{a(\sqrt{P})}{a(1)} = \frac{x+1}{x+\frac{1}{\sqrt{P}}} = 1 + \frac{1-\frac{1}{\sqrt{P}}}{x+\frac{1}{\sqrt{P}}} \geq 1 + \frac{0.99}{0.11} = 10$, which contradicts the competitive ratio of $\mathcal{A}$.
- $\mathcal{U} \subseteq [0, 10]$, otherwise there would exist an $x > 10$ such that $\mathcal{A}$ allocates less than $\sqrt{P}$ processors for $t(p) = \frac{x}{p} + \frac{1}{\sqrt{P}}$, and we would have $\frac{t}{t^{\min}} \geq \frac{t(\sqrt{P})}{t(P)} = \frac{\frac{x}{\sqrt{P}}+\frac{1}{\sqrt{P}}}{\frac{x}{P}+\frac{1}{\sqrt{P}}} > \frac{x}{\frac{x}{\sqrt{P}}+1} = \frac{1}{\frac{1}{\sqrt{P}}+\frac{1}{x}} > \frac{1}{0.11} > 9$, which also contradicts the competitive ratio $\mathcal{A}$.

Based on the previous analysis, we now consider $s = sup(\mathcal{U}) \in [0.1, 10]$. By definition, there exists a $\delta \in [0, \frac{1}{\sqrt{P}})$ such that $(s - \delta) \in \mathcal{U}$ and $(s - \delta + \frac{1}{\sqrt{P}}) \notin \mathcal{U}$. We choose such δ,

and set $\bar{w} = s - \delta > 0.09$, $t_A(p) = \frac{\bar{w}}{p} + \frac{1}{\sqrt{P}}$ and $t_B(p) = \frac{\bar{w}+\frac{1}{\sqrt{P}}}{p} + \frac{1}{\sqrt{P}}$ with $p_A < \sqrt{P}$ and $p_B \geq \sqrt{P}$. Thus, we get $t_A^* \leq \bar{w} + \frac{1}{\sqrt{P}}$ and $t_B^* = \bar{w} + \frac{2}{\sqrt{P}}$ ($R_2, R_7 : \checkmark$).

We can further assume $p_B \leq P^{3/4}$, otherwise we would have $\frac{a_B}{a_B^{\min}} \geq \frac{\bar{w}+\frac{P^{3/4}}{\sqrt{P}}}{\bar{w}+\frac{2}{\sqrt{P}}} \geq \frac{0.09+P^{1/4}}{11} > 5$, which contradicts the competitive ratio of $\mathcal{A}$ ($R_3 : \checkmark$). Finally, we set $p_A^* = \lfloor P^{3/4} \rfloor$ ($R_1 : \checkmark$) and get:

$$\begin{aligned}
\frac{a_B}{a_B^*(1-\mu^{\text{AMD}})} + \frac{t_A}{t_A^*} &\geq \frac{\bar{w}+1}{(\bar{w}+\frac{2}{\sqrt{P}})(1-\mu^{\text{AMD}})} + \frac{\frac{\bar{w}}{\sqrt{P}} + \frac{1}{\sqrt{P}}}{\frac{\bar{w}}{P^{3/4}-1} + \frac{1}{\sqrt{P}}} \\
&= \frac{\bar{w}+1}{\bar{w}(1-\mu^{\text{AMD}})} \cdot \frac{1}{1+\frac{2}{\bar{w}\sqrt{P}}} + \frac{\bar{w}+1}{1+\frac{\bar{w}}{P^{1/4}-\frac{1}{\sqrt{P}}}} \\
&\geq \frac{\bar{w}+1}{\bar{w}(1-\mu^{\text{AMD}})} \left(1 - \frac{2}{\bar{w}\sqrt{P}}\right) + (\bar{w}+1)\left(1 - \frac{\bar{w}}{P^{1/4}-\frac{1}{\sqrt{P}}}\right) \\
&\geq \frac{\bar{w}+1}{\bar{w}(1-\mu^{\text{AMD}})} + \bar{w} + 1 - \frac{22}{0.09^2 \times 0.5\sqrt{P}} - \frac{110}{P^{1/4}-\frac{1}{\sqrt{P}}} \\
&\geq \frac{\bar{w}+1}{\bar{w}(1-\mu^{\text{AMD}})} + \bar{w} + 1 - \epsilon \ .
\end{aligned}$$

We now set $x = \frac{\bar{w}+1}{\bar{w}} > 1$, so $\frac{x}{x-1} = \bar{w} + 1$. We can finally conclude that:

$$\begin{aligned}
\frac{a_B}{a_B^*(1-\mu^{\text{AMD}})} + \frac{t_A}{t_A^*} &\geq \frac{x}{1-\mu^{\text{AMD}}} + \frac{x}{x-1} - \epsilon \\
&\geq \min_{x'>1}\left(\frac{x'}{1-\mu^{\text{AMD}}} + \frac{x'}{x'-1}\right) - \epsilon \\
&= \frac{\alpha^{\text{AMD}}}{1-\mu^{\text{AMD}}} + \beta^{\text{AMD}} - \epsilon \\
&= \frac{2}{1-\sqrt{8\sqrt{2}-11}} - \epsilon \ .
\end{aligned}$$

To show the third step above, we can take the derivative of $\frac{x'}{1-\mu^{\text{AMD}}} + \frac{x'}{x'-1}$ and show that its minimum is achieved when x' satisfies $(x'-1)^2 = 1 - \mu^{\text{AMD}}$. This is equivalent to $x' = \alpha^{\text{AMD}}$, because $\left(\alpha^{\text{AMD}} - 1\right)^2 = \frac{\left(\sqrt{2}-1+\sqrt{2\sqrt{2}-1}\right)^2}{4} = \frac{1+(\sqrt{2}-1)\sqrt{2\sqrt{2}-1}}{2} = \frac{1+\sqrt{8\sqrt{2}-11}}{2} = 1 - \mu^{\text{AMD}} = (x'-1)^2$. As $x' > 1$, the only solution is $x' = \alpha^{\text{AMD}}$. From the proof of Lemma 20, we also get that $\beta^{\text{AMD}} = \frac{\alpha^{\text{AMD}}}{\alpha^{\text{AMD}}-1} = \frac{x'}{x'-1}$.

Finally, for task D, we set $t_D(p) = \frac{\epsilon}{121P^2}$ and $p_D^* = 1$. Thus, we must have $p_D \leq 4$, otherwise $\frac{a_D}{a_D^{\min}} > 4$, which contradicts the competitive ratio of $\mathcal{A}$ ($R_4, R_5, R_6 : \checkmark$). □

3.5.3 Step 2: Global Analysis

In this section, we assume that algorithm $\mathcal{A}$ and model M are fixed, while the tasks A, B, C, D and algorithm $\mathcal{A}^*$ are chosen such that the conditions of Theorem 16 hold. We construct a task graph (as shown in Figure 3.1), based on which we will show that $\frac{T}{T^*} \geq \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 2\epsilon \geq \frac{1}{\mu^M} - 3\epsilon$.

In our constructed task graph, the tasks are partitioned into four different groups: $\mathcal{T}_A, \mathcal{T}_B, \mathcal{T}_C$ and $\mathcal{T}_D$. Specifically,

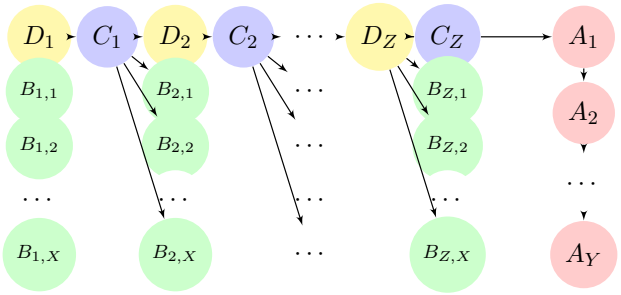


Figure 3.1: A task graph for proving lower bounds.

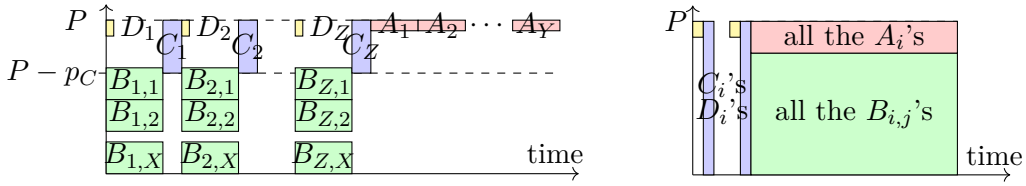


Figure 3.2: Shapes of algorithm $\mathcal{A}$'s schedule (a) and algorithm $\mathcal{A}^*$'s schedule (b) for the task graph of Figure 3.1.

- $\mathcal{T}_A$ has Y tasks identical to A, labeled as $(A_i)_{i\in[1,Y]}$;
- $\mathcal{T}_B$ has XZ tasks identical to B, labeled as $(B_{i,j})_{i\in[1,Z],j\in[1,X]}$;
- $\mathcal{T}_C$ has Z tasks identical to C, labeled as $(C_i)_{i\in[1,Z]}$;
- $\mathcal{T}_D$ has Z tasks identical to D, labeled as $(D_i)_{i\in[1,Z]}$,

where $X = \left\lceil \frac{P-p_C+1}{p_B} \right\rceil$, and using $K = \left\lceil \frac{5t_A^*}{\epsilon X t_B^*} \right\rceil$, we set $Y = \left\lfloor \frac{XKt_B^*}{t_A^*} \right\rfloor$ and $Z = K(P - p_A^*)$. These parameters are specified as definitions (F's) in Table IV.

The tasks are organized in layers and have the following precedence constraints:

- task C_i is the predecessor of task D_{i+1} for $1 \leq i < Z$, and of tasks $B_{i+1,j}$ for $1 \leq i < Z$ and $1 \leq j \leq X$;
- task D_i is the predecessor of task C_i for $1 \leq i \leq Z$;
- task C_Z is the predecessor of task A_1;
- task A_i is the predecessor of task A_{i+1} for $1 \leq i < Y$.

To prove the lower bound, we will first show that the constraints (R_{13}) to (R_{15}) in Table IV pertaining to the parameters of the task graph are also respected (Lemma 24). We will then show that algorithm $\mathcal{A}$'s schedule must follow the shape as shown in Figure 3.2(a) (Lemma 25), whereas algorithm $\mathcal{A}^*$ could wait until tasks in $\mathcal{T}_C$ and $\mathcal{T}_D$ are finished before launching tasks in $\mathcal{T}_A$ and $\mathcal{T}_B$, resulting in a better schedule as shown in Figure 3.2(b). This last result together with Theorem 16 will lead to a contradiction, hence proving the lower bound (Theorem 17). In the following analysis, we will provide a reference to a constraint or a definition whenever it is used.

Lemma 24. *Given the setting above, constraints* (R_{13}), (R_{14}) *and* (R_{15}) *in Table IV are satisfied.*

Proof. Constraint (R_{13}) can be obtained directly from the definition of X:

$$1 \leq X = \left\lceil \frac{P - p_C + 1}{p_B} \right\rceil \leq P \ . \qquad (F_2)$$

Constraint (R_{14}) can be derived from the definitions of Y and K:

$$\begin{aligned} XKt_B^* \geq Yt_A^* &\geq XKt_B^* - t_A^* && (F_4) \\ &= XKt_B^* \left(1 - \frac{t_A^*}{XKt_B^*}\right) \\ &\geq XKt_B^* \left(1 - \frac{\epsilon}{5}\right) . && (F_3) \end{aligned}$$

Finally, constraint (R_{15}) can be obtained with:

$$\begin{aligned} K(P - P^{3/4}) \leq Z &\leq KP && (F_5, R_1) \\ &\leq \left(\frac{5t_A^*}{\epsilon Xt_B^*} + 1\right) P && (F_3) \\ &\leq \left(\frac{120}{\epsilon} + 1\right) P && (R_7, R_{13}) \\ &\leq \frac{121P}{\epsilon} . && (\epsilon < 1) \end{aligned}$$

□

Lemma 25. *For a given task $\mathcal{T}$, let $s(\mathcal{T})$ denote its starting time in algorithm $\mathcal{A}$'s schedule and $e(\mathcal{T})$ its ending time. If all constraints in Table IV are satisfied, then algorithm $\mathcal{A}$'s schedule must follow the shape as shown in Figure 3.2(a), i.e.:*

- $s(D_i) = s(B_{i,1}) = \cdots = s(B_{i,X}), \forall i \in [1, Z]$;
- $s(C_i) = e(B_{i,1}) = \cdots = e(B_{i,X}), \forall i \in [1, Z]$;
- $s(D_i) = e(C_{i-1}), \forall i \in [2, Z]$;
- $s(A_1) = e(C_Z)$;
- $s(A_i) = e(A_{i-1}), \forall i \in [2, Y]$.

As a result, the makespan of $\mathcal{A}$ must satisfy: $T \geq Zt_B + Yt_A$.

Proof. It is possible to simultaneously run all the tasks $B_{i,j}$'s and D_i in layer i, as the total number of processors required is:

$$\begin{aligned} Xp_B + p_D &\leq \left(\frac{P - p_C + 1}{p_B} + 1\right) P_B + 4 && (F_2, R_6) \\ &= P - P_C + P_B + 5 \\ &\leq (1 - \mu^M)P + P^{3/4} + 5 && (R_3, R_{12}) \\ &< 0.8P + 0.01P + 5 < P . && (F_1, \mu_M > 0.2) \end{aligned}$$

However, it is not possible to run all the $B_{i,j}$'s and C_i in parallel, as the number of processors required would be:

$$\begin{aligned} Xp_B + p_C &\geq \frac{P - p_C + 1}{p_B} p_B + p_C && (F_2) \\ &= P + 1 . \end{aligned}$$

Therefore, given that algorithm $\mathcal{A}$ uses list scheduling to schedule the tasks, we get $s(D_1) = s(B_{1,1}) = \cdots = s(B_{1,X})$. Furthermore, since $t_D \leq t_B$ (R_4), C_1 becomes available before the first layer of tasks in $\mathcal{T}_B$ finishes and will be launched as soon as the layer is done, which gives $s(C_1) = e(B_{1,1}) = \cdots = e(B_{1,X})$. A direct induction shows that, using list scheduling, the same scenario would happen for all the Z layers. Finally, the tasks in $\mathcal{T}_A$ are executed one after another after the completion of C_Z, so the schedule corresponds to the exactly one shown in Figure 3.2(a). Since there are Z layers of tasks in $\mathcal{T}_B$ and Y layers of tasks in $\mathcal{T}_A$, the makespan of algorithm $\mathcal{A}$ must satisfy $T \geq Zt_B + Yt_A$. □

Theorem 17. *Given a model $M \in \{\textsc{Roo}, \textsc{Com}, \textsc{Amd}\}$, there is no online list scheduling algorithm respecting Definition 2 with a competitive ratio strictly less than $\frac{1}{\mu^M}$.*

Proof. We prove the theorem by contradiction. Specifically, we assume that there exists such an algorithm $\mathcal{A}$ with a competitive ratio of $\frac{1}{\mu^M} - 4\epsilon$ for some $0 < \epsilon < 1$, as also assumed in Theorem 16. We will then show, using the constructed task graph, that the makespan of $\mathcal{A}$ satisfies $\frac{T}{T^*} \geq \frac{1}{\mu^M} - 3\epsilon$, which leads to a contradiction, hence suggesting that no such algorithm should exist.

We first bound the makespan T^* of algorithm $\mathcal{A}^*$, assuming that it follows the schedule of Figure 3.2(b) by first running all the C_i's and D_i's before running the A_i's sequentially while at the same time using one processor to run each of the $B_{i,j}$'s. As there are XZ tasks of $B_{i,j}$'s, executing them all on $P - p_A^*$ processors takes time $\left\lceil \frac{XZ}{P-p_A^*} \right\rceil t_B^* = \left\lceil \frac{XK(P-p_A^*)}{P-p_A^*} \right\rceil t_B^* = XKt_B^*$ (F_5), whereas executing all the A_i's takes time $Yt_A^* \leq XKt_B^*$ (R_{14}). Therefore, T^* should satisfy:

$$\begin{aligned} T^* &\leq Z(t_C^* + t_D^*) + XKt_B^* \\ &\leq \frac{121P}{\epsilon}\left(\frac{\epsilon}{121P^2} + \frac{\epsilon}{121P^2}\right) + XKt_B^* \qquad (R_5, R_{11}, R_{15}) \\ &\leq \frac{2}{P} + XKt_B^* \ . \end{aligned}$$

Now, using the result of Lemma 25, we get:

$$\begin{aligned} \frac{T}{T^*} &\geq \frac{Zt_B + Yt_A}{\frac{2}{P} + XKt_B^*} \\ &\geq \frac{K(P - P^{3/4})t_B}{2 + XKt_B^*} + \frac{Yt_A}{\frac{2}{P} + Yt_A^*/\left(1 - \frac{\epsilon}{5}\right)} \qquad (R_{14}, R_{15}) \\ &= \frac{(P - P^{3/4})t_B}{\frac{2}{K} + Xt_B^*} + \frac{t_A}{t_A^*/\left(1 - \frac{\epsilon}{5}\right) + \frac{2}{YP}} \\ &\geq \frac{(P - P^{3/4})t_B}{2 + t_B^*\left(\frac{P(1-\mu^M)+1}{p_B} + 1\right)} + \frac{t_A}{t_A^*} \cdot \frac{1 - \frac{\epsilon}{5}}{1 + \frac{2}{PXKt_B^*}(1 - \frac{\epsilon}{5})} \qquad (F_2, R_{12}, F_4) \\ &\geq \frac{(P - P^{3/4})t_B}{2 + 2t_B^* + \frac{Pt_B^*(1-\mu^M)}{p_B}} + \frac{t_A}{t_A^*} \cdot \frac{1 - \frac{\epsilon}{5}}{1 + \frac{2}{Pt_B^*}} \\ &\geq \frac{t_B p_B}{\frac{p_B(2+2t_B^*)}{P - P^{3/4}} + \frac{Pt_B^*(1-\mu^M)}{P - P^{3/4}}} + \frac{t_A}{t_A^*} \cdot \frac{1 - \frac{\epsilon}{5}}{1 + \frac{20}{P}} \qquad (R_2) \\ &\geq \frac{a_B}{\frac{2+2t_B^*}{P^{1/4}-1} + \frac{t_B^*(1-\mu^M)}{1-P^{-1/4}}} + \frac{t_A}{t_A^*} \cdot \frac{1 - \frac{\epsilon}{5}}{1 + \frac{20}{P}} \qquad (R_3) \\ &\geq \frac{a_B}{(4 + 4t_B^*)P^{-1/4} + t_B^*(1 - \mu^M)(1 + 2P^{-1/4})} + \frac{t_A}{t_A^*} \cdot \frac{1 - \frac{\epsilon}{5}}{1 + \frac{20}{P}} \ . \end{aligned}$$

The last step above assumes $\frac{1}{P^{1/4}-1} \leq 2P^{-1/4}$ and $\frac{1}{1-P^{-1/4}} \leq 1 + 2P^{-1/4}$, both of which

are true if $P^{1/4} \geq 2$, i.e., $P \geq 16$. We conclude with the following derivations:

$$
\begin{aligned}
\frac{T}{T^*} &\geq \frac{a_B}{t_B^*(1-\mu^M)+(4+6t_B^*)P^{-1/4}} + \frac{t_A}{t_A^*} \cdot \frac{1-\frac{\epsilon}{5}}{1+\frac{20}{P}} \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)+604P^{-1/4}} + \frac{t_A}{t_A^*} \cdot \frac{1-\frac{\epsilon}{5}}{1+\frac{20}{P}} && (R_2, R_8) \\
&= \frac{a_B}{a_B^*(1-\mu^M)} \cdot \frac{1}{1+\frac{604P^{-1/4}}{a_B^*(1-\mu^M)}} + \frac{t_A}{t_A^*} \cdot \frac{1-\frac{\epsilon}{5}}{1+\frac{20}{P}} \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)} \cdot \frac{1}{1+12080P^{-1/4}} + \frac{t_A}{t_A^*} \cdot \frac{1-\frac{\epsilon}{5}}{1+\frac{20}{P}} && (R_2, \mu^M \leq 0.5) \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)} \left(1-12080P^{-1/4}\right) + \frac{t_A}{t_A^*} \left(1-\frac{20}{P}\right)\left(1-\frac{\epsilon}{5}\right) \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 120800P^{-1/4} - \frac{100}{P} - \epsilon && (R_9, R_{10}, \mu^M \leq 0.5) \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 120800P^{-1/4} - 100P^{-1/4} - \epsilon \\
&= \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 120900P^{-1/4} - \epsilon \\
&\geq \frac{a_B}{a_B^*(1-\mu^M)} + \frac{t_A}{t_A^*} - 2\epsilon && (F_1) \\
&\geq \frac{1}{\mu^M} - 3\epsilon \ .
\end{aligned}
$$

The last step above applies Theorem 16, and the result proves this theorem and the optimal competitive ratio of our algorithm for these speedup models. □

Remarks. Since the Amdahl's model is a special case of the mix model, its lower bound also applies to the mix model.

3.6 Conclusion and Future Work

In this chapter, we have studied the online scheduling of moldable task graphs to minimize makespan with tasks obeying several common speedup models. To the best of our knowledge, no competitive ratio was known under this setting, except for the roofline model [60]. Owing to the design of a new online algorithm and a novel analysis framework, we have extended the result and derived competitive ratios for several other speedup models, including the communication model, the Amdahl's model and a mix combination. We have also shown that no online list scheduling algorithm with deterministic local decisions for processor allocation may have a better competitive ratio than ours for the roofline, communication and Amdahl's models. Finally, we have considered the arbitrary speedup model and established a lower bound for any deterministic online algorithm. Altogether, these new results lay the foundations for further study of this important but difficult scheduling problem.

For future work, we will consider extending the algorithm and analysis to other common speedup models. We also plan to extend our algorithm and analysis to other online scheduling settings (e.g., for independent tasks released over time, and for special task graphs such as fork-join graphs or trees). Finally, we will expand this study to a more practical side by experimentally benchmarking the performance of our algorithm using realistic workflows.

Chapter 4

Risk-Aware Scheduling Algorithms for Variable Capacity Resources

In Chapters 2 and 3, we have studied scheduling problems where the jobs (tasks) were subject to failures, and we have generalized these results to the online scheduling problem. In this chapter, we rather focus on the resilience when the variability is induced by the platform instead of the jobs. Indeed, the drive to decarbonize the power grid to slow the pace of climate change has caused dramatic variation in the cost, availability, and carbon-intensity of power. This has begun to shape the planning and operation of datacenters. This chapter focuses on the design of scheduling algorithms for independent jobs that are submitted to a platform whose resource capacity varies over time. Jobs are submitted online and assigned on a target machine by the scheduler, which is agnostic to the rate and amount of resource variation. The optimization objective is the goodput, defined as the fraction of time devoted to effective computations (re-execution does not count). We introduce several novel algorithms that: (i) decide which fraction of the resources can be used safely; (ii) maintain a risk index associated to each machine; and (iii) achieves a global load balance while mapping longer jobs to safer machines. We assess the performance of these algorithms using one set of actual workflow traces together with three sets of synthetic traces. The goodput achieved by our algorithms increases up to 10% compared to standard first-fit approaches, while we never experience any loss in complementary metrics such as the maximum or average stretch. This chapter corresponds to Submission [S4] (see Chapter 9).

4.1 Introduction

With growing global concern about climate change [161, 162], the end of Dennard scaling, and continued exponential growth of computing use [98, 152], there is growing interest in how to reduce the negative environmental impacts of computing, that is, improve its sustainability [63, 75, 135, 168]. A popular approach that exploits the variation of renewable generation (wind, solar), is the idea of temporal and spatial load shifting to match computing power consumption with the availability of low-carbon power [72]. Viewed at a single computing site, this appears as a datacenter or HPC platform whose capacity evolves with time, depending upon the cost and the environmental policy (e.g., reduce power supply when the power source is coal instead of wind or solar), termed the *variable capacity scheduling* problem [177].

Online scheduling techniques for independent jobs have received considerable attention since they lie at the heart of *batch schedulers*. The traditional setting deals with fixed-

$\mathcal{J}$	Set of jobs
n	Total number of jobs
τ_i	Represents a job to execute
r_i	Release date of job τ_i
c_i	Number of cores required to process job τ_i
w_i	Length of job τ_i
C_i	Category of job τ_i (based on its length)
T_{avg}	Average job length
n_c	Number of cores per machine
M^+	Total number of machines
M^-	Number of machines always available
M_{ra}	Range of the machines ($M_{avg} = M^+ - M_{ra} = M^- - M_{ra}$)
M_{avg}	Average number of machines
M_{use}	Current number of machines usable by our heuristics
D^+	Maximum distance between the target machine (based on C_i) and the machine chosen by our heuristic
$u(m)$	Utilization of a machine
U	Total utilization of machines
ϕ	The number of machines change every ϕ seconds
S^+	Maximum Stretch of the jobs processed so far
$[T_{begin}, T_{end}]$	Steady-state window

Table I: Summary of main notations for Chapter 4.

capacity resources that do not evolve over time, such as a parallel HPC platform. In this standard setting, there are many optimization objectives. From the platform owner's perspective, the standard objective is to maximize utilization, defined as the fraction of time where platform resources execute computations [133]. From the user's perspective, the standard objective is to minimize the maximum (or sometimes average) stretch. The stretch of a job is the ratio of its response time (time elapsed between submission and completion) over its execution time, and is preferred to the plain response time for fairness [19, 40].

This chapter focuses on scheduling techniques for independent jobs, when variations in power supply imply changes in the number of available computing resources over time. The scheduler is agnostic to the rate and amount of resource variations but must prepare for such variations. In particular, a given resource may be shut down abruptly because of a power shortage, in which case all the jobs executing on that resource are interrupted and must be re-executed later on. Therefore, we design risk-aware strategies that assign incoming jobs to the *right* target machine, with some optimization criteria in mind. Because today's power grids have rapid (hourly, daily, weekly) and large variation (3-5x today, growing to 10x), and shifting benefit increases with the magnitude of load shifted, we consider dynamic ranges as large as 80% of the maximum capacity. With varying resources, platform utilization is no longer an adequate criterion, because partial executions of jobs that get interrupted do not count as actual progress of the jobs. We use the goodput instead to account for this.

This chapter lays the foundation for a risk-aware strategy that maps jobs onto machines. We make several simplifying assumptions that enable us to design a model and assess the efficiency of our algorithms by simulation:

- The target platform consists of a collection of identical parallel machines, each equipped with many cores;

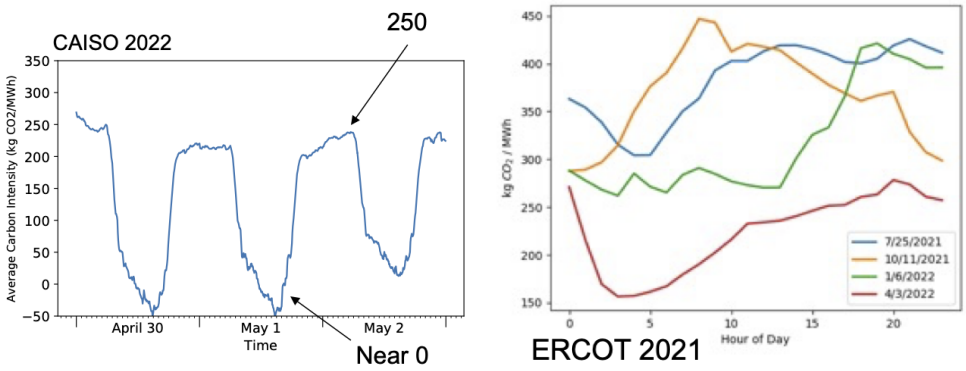


Figure 4.1: Daily CAISO Variation of CO2/MWh is 5x and can exceed 100x (CAISO, 2022); seasonal variation (carbon/MWh) is 3x and growing.

- Variations in power capacity imply changes in the number of machines that can be alive at a given time t Alive machines at time t are defined as machines that are switched on and execute jobs at time t;
- Power consumption at time t is directly proportional to the number of machines that are switched on at time t; thereby we ignore possible variations due to the actual load, operating frequency and application behavior on each machine at time t;
- The number of alive machines at time t is not known before time t. Instead, it obeys a random walk that evolves from the number of alive machines at time $t-1$;

Maybe the most drastic assumption is the last one: machines are added to and removed from the pool of available machines without explicit warning. It implies that jobs that are running on an alive machine get interrupted without notice when that machine gets removed from the pool. One can envision several alternative approaches, such as:

1. Execution can continue when the machine is removed from the pool, but at a higher price. This would model the scenario where alive machines are powered by green sources, and brown power can complement green power when needed.
2. Variations of capacity are known some time in advance, so that jobs running on machines that are going to be switched off soon could take a proactive checkpoint and then continue on another machine from the checkpointed state, instead of resuming execution from scratch.

The first approach requires to use total cost as a complementary objective, and a model to take it into account. The second approach requires many additional parameters, both for the prediction mechanism (prediction time, recall and precision) and for job characteristics (checkpointable or not, checkpoint durations). Instead, we rely on a simple but reasonable Markov model for machine availability, which has been shown accurate to model resource variation in several frameworks [164]. Variations in wind power nicely obey such a Markov model [134]. Variations in solar power would require a more complicated model, such as a heterogeneous Markov chain to account for different contexts (e.g., day or night) [175].

Our main contributions are the following:

1. We provide a simplified yet realistic model for power variation that translates into the number of available machines obeying a random walk at each step, and we present the first complexity results for the *variable capacity scheduling* problem
2. We design novel risk-aware scheduling and mapping algorithms that are capable of mitigating the impact of power variation by using several new techniques, such as:

(i) keeping an ordered list of machines ordered by potential risk; (ii) mapping each job to a given set of machines according to its relative length w.r.t. the set of jobs released so far; (iii) maintaining local queues to achieve a bounded maximum stretch; (iv) re-execute interrupted jobs on new machines when power supply decreases; and (v) re-distribute pending jobs on new machines when power supply increases

3. We assess the performance of these novel algorithms using an extended set of simulations, and report significant gains in the achieved goodput over standard first-fit algorithms. Nicely, this increase of the goodput is achieved without any loss in complementary metrics such as maximum or average stretch.

The rest of the chapter is organized as follows. Section 4.2 surveys related work. In Section 4.3, we provide the framework for platforms, jobs, resource variability and optimization objectives. Section 4.4 presents complexity results. In Section 4.5, we introduce our novel scheduling and resource assignment algorithms, whose experimental assessment is conducted in Section 4.6. Finally, we give concluding remarks and hints for future work in Section 4.7.

4.2 Related Work

Prior research has explored resource management for sustainability from many different perspectives. Several surveys [33, 114, 117, 118, 131] detailed current energy awareness and power management techniques that are regularly applied in HPC centers. In [117], the authors summarize a set of recent efforts to study new techniques by a collection of HPC centers around the world. Notably, those include:

- experiments to evaluate energy and costs savings by allowing the job scheduler to inform the selection of the power source at RIKEN Japan (the alternative energy sources are limited to the electrical grid and local production of electricity via gas turbines);
- researchers at Tokyo Tech Japan study the possibility to shut down idle nodes, by coupling the job scheduler decision with the resource manager;
- some centers (CEA France, STFC UK, LANL USA, CINECA Italy) evaluate technologies to couple performance metric gathering and job scheduling, in order to report energy usage to the user, or build statistical databases of energy usage to inform future decisions;
- other centers are experimenting with monitoring and power capping tools to manage power usage and adapt it to variable external conditions (KAUST Saudi Arabia, LRZ Germany).

While most research efforts on energy efficiency focus on fixed resource capacities, a body of research that deals with the addition and removal of resources with time has continued to grow. [167] considers the case of volatile computing resources that are turned fully on or off to best utilize stranded renewable generation. Live migration of VMs between hosts as an energy efficiency mechanism is explored in [18]. The scheduler would continuously consolidate VMs and migrate between hosts, while respecting Quality of Service (QoS) requirements. Energy saving are achieved by switching off idle physical nodes. [177] defines a framework of dimensions for variation, including dynamic range, frequency of change, and shape.

Shut-down models, where a system is put into a sleep state when idle, are also shown to reduce energy consumption, and provide energy-saving algorithms that execute in polynomial time [12]. Speed scaling mechanisms, such as Dynamic Voltage and Frequency Scaling (DVFS), allow processors and servers to run at lower speed at the cost of increased

execution times. [165] provides a scheduling algorithm that uses DVFS to allocate jobs in cloud datacenters, reducing energy consumption.

Past studies have explored scheduling policies that assure secure job execution in the presence of unpredictable resource failures. [145] extends the known scheduling heuristics, preemptive, replication and delay-tolerant, to provide security assurances. [146] constructs statistical models to assess the reliability of resources based on prior performance and behavior, considering this reputation-based reliability rating in the job allocation algorithm. Reputation-based scheduling on unreliable resources [11, 146] can be considered parallel to our work. However, this body of work deals with assigning jobs to redundancy groups, while our work relates to scheduling independent jobs on volatile nodes.

Prediction of renewable generation has been considered in providing insights to job scheduling for energy efficiency. GreenSlot [68] is a parallel batch job scheduler for a datacenter powered by solar energy and connected to the electrical grid as a backup. GreenSlot predicts near-term solar energy generation and schedules the workload to maximize the green energy consumption while meeting the jobs' deadlines based on slack. Similarly, [2] utilizes short-term prediction of both solar and wind energy generation to improve green energy utilization, while reducing the number of terminated jobs.

Opportunities to mitigate performance degradation from capacity variation through intelligent termination of jobs are explored in [177]. Two policies that terminate jobs such that wasted work is minimized or terminated jobs with the least fraction completed are evaluated. While [177] takes a reactive approach on handling variable resource capacity, our work takes a proactive approach in scheduling jobs on these resources, and provides novel algorithms to assign jobs to machines. Scheduling on computing resources provisioned with 100% renewable energy is further analyzed in [101]. Greedy and binary search scheduling algorithms are evaluated as heuristics for minimizing the makespan and flowtime without job preemption.

None of the aforementioned work directly addresses the problem of online risk-aware scheduling on variable capacity resources. However, they provide an important backdrop to techniques that could potentially be adapted for this purpose. We build upon widely accepted heuristics, assigning jobs to individual physical nodes in a highly volatile renewable based environment.

4.3 Framework

This section details the framework and the objective function.

4.3.1 Target Platform

We consider a parallel platform that consists of a set $\mathcal{M}$ of M^+ identical parallel machines, each equipped with N cores. Each machine $m \in \mathcal{M}$ requires a certain amount of power $\hat{P}$ to run. For simplicity, we assume that $\hat{P}$ is constant whenever the machine is switched on, even if some of the cores are unused. In other words, $\hat{P}$ is proportional to the number of cores N available in machine m, i.e. $\hat{P} = p \times N$ for some constant p. Our scheduling problem includes capacity variations, where the overall available power capacity is a function of time t, denoted as $P(t)$. We discretize time and assume that t takes integer values expressed in the appropriate unit, e.g., seconds. We never need a power exceeding $\hat{P}M^+$, so we can safely assume that $P(t) \leq \hat{P}M^+$ at any time t. We use $b_{m,t}$ as a boolean decision variable which is equal to 1 if machine m is active at time t and 0 otherwise. Then, at any time t,

the total number of resources used by all machines must remain below $P(t)$, i.e.

$$\forall t, \sum_{m \in \mathcal{M}} b_{m,t} \times \hat{P} \leq P(t). \tag{4.1}$$

The scheduling strategies described in this chapter easily extend to a collection of heterogeneous machines with different number of cores, and more generally different hardware characteristics. However, we do not have log traces to simulate such heterogeneous clusters.

4.3.2 Jobs

We schedule a set of independent jobs $\mathcal{J}$ on the $\mathcal{M}$ parallel machines. Each job $\tau_i \in \mathcal{J}$ is released at date r_i, needs c_i cores for execution, and has length w_i. We allocate each job τ_i to a machine m_i at a starting date s_i. We use e_i as the (predicted) completion date of job τ_i. If job τ_i is not interrupted, we have $e_i = s_i + w_i$. We let $b_{i,m,t}$ be the boolean decision variable which is equal to 1 if job τ_i is running on machine m at time t, and is equal to 0 otherwise. More formally,

$$b_{i,m,t} = 1 \Leftrightarrow (m_i = m \text{ and } s_i \leq t < e_i)\,.$$

We note that a machine m is on at time t if and only if one job is executing:

$$b_{m,t} = 1 \Leftrightarrow \sum_{\tau_i \in \mathcal{J}} b_{i,m,t} > 0$$

As already mentioned, it might happen that a job τ_i needs to be interrupted, see Section 4.3.3. In this case, we let $s_i = e_i = 0$ and re-update these values accordingly whenever the job is rescheduled.

The goal is to schedule all jobs on the machines, given the cores available on each machine. More precisely, if $t \in \mathcal{T}$ denotes any time of the whole execution (all jobs), the schedule must verify:

$$\forall m \in \mathcal{M}, \forall t \in \mathcal{T}, \sum_{\tau_i \in \mathcal{J}} b_{i,m,t} c_i \leq N; \tag{4.2}$$

$$\forall \tau_i \in \mathcal{J}, s_i \geq r_i. \tag{4.3}$$

Equation (4.2) expresses the constraint on the number of cores on each machine, while Equation (4.3) simply states that execution cannot start before release time. For simplicity, we assume that all times t take integer values (seconds).

4.3.3 Variable Resources

At a given time, a low power capacity may impose to turn some machines off. The jobs currently executing on these machines have to be interrupted immediately, and rescheduled at a further time. Of course, all previous constraints must still be enforced for re-execution. Because we target identical parallel machines, we directly consider the variation in the number of available machines $M_{alive}(t)$ instead of the power variation, using $M_{alive}(t) = \left\lfloor \frac{P(t)}{pN} \right\rfloor$.

To simulate resource variations, we use a bounded random walk for the number of machines. This walk is defined by a lower bound M^- and a higher bound M^+ on the number of machines, a variation period ϕ, and a variation step *step* on the number of machines. Every ϕ time units, the number of machines can decrease or increase by *step* or remain the same, while respecting the constraints that $M_{alive}(t) \in [M^-, M^+]$.

4.3.4 Objective Function

As already mentioned, the objective is to optimize the goodput [13, 177], which measures useful platform utilization by accounting only for jobs that have been completed, so that re-execution does not count. For any time T, we say that job $\tau_i \in \mathcal{J}_{comp,T}$ if $e_i \leq T$. Hence $\mathcal{J}_{comp,T}$ is the set of jobs that are complete at time T. Similarly, we say $\tau_i \in \mathcal{J}_{started,T}$ if $s \leq T < e_i$ (and τ_i is not dead at time T). At any time $t \in [0, T-1]$, the maximal number of cores that may be turned on is at most $M_{alive}(t)N$, otherwise the number of machines turned on would require a power larger than $P(t)$. Therefore, the total number of units of work that can be executed in $[0, T]$ is at most $\sum_{t\in[0,T-1]} M_{alive}(t)N$, and we define $\textsc{Goodput}(T)$, the goodput at time T, as the fraction of useful work up to time T:

$$\textsc{Goodput}(T) = \frac{\sum_{\tau_i\in\mathcal{J}_{comp,T}} w_i c_i + \sum_{\tau_i\in\mathcal{J}_{started,T}} (T - s_i)c_i}{N \sum_{t\in[0,T-1]} M_{alive}(t)}. \tag{4.4}$$

4.4 Complexity

In this section, we give two complexity results that show that resource variation dramatically complicates the online scheduling problem. First, we consider a very simple problem instance with only one machine, and we show that an adversary can force any scheduling algorithm to err and achieve no goodput at all :

Theorem 18. *An adversary can force any schedule to achieve no goodput at all, even with a single unicore machine.*

Proof. Consider a single unicore machine. Initially, job τ_1 of size $c_1 = 1$ and duration $w_1 = 3$ is released at time $t = r_1 = 0$. We consider the goodput of the machine at time $T = 3$. Recall that time takes integer values.

If the scheduler starts the execution of τ_1 at time $s_1 > 0$, the adversary shutdowns the machine at time $t = 2$. Then at time $T = 3$ the goodput is $\textsc{Goodput}(T) = \textsc{Goodput}(3) = 0$ because the job has not completed,. The optimal solution was to start the job immediately at time $s_1 = 0$ and achieve $\textsc{Goodput}(3) = 3$.

If the scheduler starts the execution of τ_1 at time $s_1 = 0$, the adversary releases a second job τ_2 of size $c_1 = 1$ and length $w_2 = 1$ at time $r_2 = 1$. Then the adversary shutdowns the machine at time $t = 2$. At time $T = 3$, job τ_1 has not completed and the goodput is $\textsc{Goodput}(3) = 0$. The optimal solution was to ignore job τ_1 and start τ_2 at time $s_2 = 1$, thereby achieving a goodput $\textsc{Goodput}(3) = \textsc{Goodput}(2) = 1$.

In both cases, the goodput of the scheduler is 0, and infinitely worse than the optimal goodput. □

Now, we introduce another problem instance with parallel machines and make a digression to show that the makespan of any polynomial scheduling algorithm can be arbitrarily larger than the optimal makespan. The makespan is defined as the completion time of the last job. The problem instance that we target here is completely offline: it assumes that all job release times and characteristics, as well as power variations over time, are known to the scheduling algorithm in advance, before the beginning of the execution. This strong result does not translate to the goodput directly, but demonstrates the intrinsic difficulty of the problem:

Theorem 19. *Unless $P = NP$, there is no constant polynomial-time approximation algorithm of the makespan, for the offline instance of the problem with parallel unicore machines.*

Proof. Given any constant $K > 0$, assume by contradiction that there is exists a K approximation algorithm $\mathcal{A}$ for the offline problem with two unicore machines. Consider an arbitrary instance $\mathcal{I}_1$ of 3-PARTITION-K, which is a variation 3-PARTITION, a well-known NP-complete problem in the strong sense [66]: given $3n$ strictly positive integers $a_1, a_2, \ldots, a_{3n}$ of sum nB, where $\frac{B}{4} < a_i < \frac{B}{2}$ for all i, can we find a partition of these $3n$ integers into n triplets each with sum B? The difference with 3-PARTITION is that we add the condition $n > K$ in 3-PARTITION-K. Obviously, restricting to large instances does not change the difficulty of the problem, and it is easy to see that 3-PARTITION-K remains NP-complete in the strong sense. We can encode the instance $\mathcal{I}_1$ in unary due to the strong completeness of the problem and let $size(\mathcal{I}_1) = O(n + B)$.

We construct an instance $\mathcal{I}_2$ of the offline problem which is defined as follows: (i) we have n unicore machines; (ii) $3n$ unicore jobs $\tau_1, \tau_2, \ldots, \tau_{3n}$ are released at time 0. The length of τ_i is $w_i = a_i$ for all i. The number of alive machines at any time is the following:

- all n machines are alive up to time B: for $0 \leq t < B$, $M_{alive}(t) = n$
- then no machine is available during KB seconds: $M_{alive}(B + t) = 0$ for $0 \leq t < KB$
- all n machines are alive afterwards: $M_{alive}(B + KB + t) = n$ for $0 \leq t < nB$

The size of $\mathcal{I}_2$ is $size(\mathcal{I}_2) = O(n + B + K) = O(n + B)$ since $K < n$, hence the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$.

We use $\mathcal{A}$ for $\mathcal{I}_2$ to determine whether $\mathcal{I}_1$ has a solution or not, thereby showing that $P = NP$, the desired contradiction.

First if $\mathcal{I}_1$ has a solution, the optimal scheduling will complete all jobs at time B, by perfectly balancing jobs across machines using the n triplets of the solution of $\mathcal{I}_1$. Because $\mathcal{A}$ is a K-approximation, it will return a makespan not exceeding K times the optimal, hence KB.

But if $\mathcal{I}_1$ does not have a solution, one cannot complete all jobs by time B. Because $M_{alive}(B) = 0$, at least one job will need to be re-executed later. But no machine is available until time $B + KB$. Hence the optimal makespan, and a fortiori the makespan returned by $\mathcal{A}$, will be at least $(1 + K)B > KB$. We can indeed use the makespan returned by $\mathcal{A}$ to decide whether $\mathcal{I}_1$ has a solution or not. □

4.5 Algorithms

In this section, we describe various scheduling algorithms for our problem with varying number of machines. Each algorithm will be defined by its actions on the following four key events that occur during execution: job arrival, job completion, machine addition, and machine removal. More specifically:

- **Job Arrival Event**: When a job is released, a decision must be made to decide when to schedule it and on which machine.
- **Job Completion Event**: When a job is completed, the cores it was using are released, possibly allowing for additional jobs to be scheduled.
- **Machine Addition Event**: When a new machine becomes available, a decision must be made on how to utilize it.
- **Machine Removal Event**: When a machine has to be turned off, jobs currently executing on that machine are killed, and a decision must be made on how to reallocate them.

Defining a strategy for each of these events fully describes the algorithm's decision-making process in response to changes in resource availability. In the following, each algorithm will be presented in a dedicated subsection, with a detailed description of its actions on each event.

4.5.1 FirstFitAware

In this section, we present a baseline heuristic against which we will compare our own scheduling algorithms. The heuristic is called FirstFitAware, labels the machines from 1 to M^+, and schedules jobs on the machine with the smallest available index that has enough free resources to execute it. Similarly, when a machine needs to be removed, FirstFitAware kills the machine with the highest index. We describe the actions of FirstFitAware on each of the four key events:

- **Job Arrival Event**: For each incoming job, FirstFitAware assigns it to the machine with the smallest available index that has enough free resources to execute it. If no machine can execute the job, it is placed in a waiting queue.
- **Job Completion Event**: When a job is completed, FirstFitAware checks the queue for the job with the smallest release date that fits in the machine where the job was completed, and assigns it to this machine. If there are no jobs in the queue or if the machine does not have enough cores available to process any of the waiting jobs, no action is taken. If a job is assigned, FirstFitAware continues to search the queue for additional jobs that can be assigned to the same machine until none fits.
- **Machine Addition Event**: When a machine is added, FirstFitAware examines the queue and assigns jobs to the new machine in order of increasing release date until no further jobs can be assigned or the queue is empty.
- **Machine Removal Event**: When a machine must be removed, FirstFitAware shuts down the machine with the highest index and places all jobs running on that machine in the queue. It then examines the queue and assigns jobs to available machines in order of increasing release date until no further jobs can be assigned or the queue is empty.

FirstFitAware is risk-aware in the sense it maintains an ordered list of machines from left (small indices) to right (large indices). Jobs are mapped to leftmost machines whenever possible, and rightmost machines are those that are shutdown whenever necessary.

4.5.2 FirstFitUnaware

For the sake of comprehensiveness, we also compare our algorithms to a second baseline heuristic, FirstFitUnaware. This second heuristic is identical to FirstFitAware except that it selects a random machine to be removed instead of always choosing the machine with the highest index. More precisely, while FirstFitAware obeys the permutation $(M^+, M^+ - 1, \cdots, 1)$ for the order of extinction (larger machine index first), FirstFitUnaware applies a random permutation. This means that FirstFitUnaware does not give priority to machines that are less likely to be removed, so we say it is not aware of the risk of shutdown incurred by the machines. Comparing both variants will help assess whether a very simple risk-aware approach is more efficient than a traditional approach unaware of power variation.

4.5.3 TargetStretch

In this section, we present TargetStretch, the first of our novel algorithms. The rationale for TargetStretch is as follows: even though FirstFitAware gives priority to machines with lower indices when assigning jobs, jobs running on machines with higher indices may have been running for a long time, and their interruption could result in significant work loss. To address this limitation, we schedule smaller jobs on machines that are likely to be turned off after some time; and longer jobs on machines that will never be

turned off. Like FirstFitAware, we always turn off the machine with the highest index. However, for job assignment, we make decisions based on job lengths. This requires several new concepts:

- Unlike before, instead of using a single queue for all machines, we have one queue per machine. Specifically, when a job arrives, we choose a machine for it and schedule it on that machine. Specifically, if the job can start immediately on the machine, we proceed and remember its (expected) end date. Otherwise, we plan for the earliest possible start date for the job, which corresponds to the smallest start-finish interval that contains enough cores to schedule our job. At each time t, all jobs planned for this machine have an exact predicted start date s_i and end date e_i. If the machine has one or more jobs planned but not yet started (for which $s_i > t$), we say its utilization $u(m)$ is 1. Otherwise, its utilization is the proportion of active cores $u(m) = \frac{c_m^A}{N}$. Here c_m^A denotes the total number of cores used by all the jobs running at time t
- In addition, we define the number of usable, i.e., risk-free, machines at any given time, M_{use}, as follows. Initially, we set $M_{use} = M^-$. Thus, when a job is assigned to a machine with an index between 1 and M_{use}, we take no initial risk. However, it would be a waste not to use the more risky machines at all, if the current set of jobs cannot fit on the risk-free machines. For this reason, we update the number of usable machines based on the utilization of the machines U defined as $U = \frac{\sum_{m\in[1,M_{use}]} u(m)}{M_{use}}$. Whenever we have $U > 0.95$, if the number of active machines $M_{alive}(t)$ is greater than M_{use}, we increase M_{use}. Conversely, if U drops below 0.8, we decrease M_{use} without interrupting the jobs running on the machines whose index is larger than M_{use}. This allows us to avoid taking too many risks unnecessarily, while using all machines when needed.
- Finally, we calculate for each job τ_i its category C_i which is based upon its relative area (defines as $w_i \times c_i$) with respect to a set $\mathcal{J}'$ of other jobs, as follows:

$$C_i = \frac{\sum_{k\in\mathcal{J}', w_k \geq w_i} w_k c_k}{\sum_{k\in\mathcal{J}'} w_k c_k} \tag{4.5}$$

  Here, the set of jobs $\mathcal{J}'$ is chosen as a set of jobs that resemble the set of jobs $\mathcal{J}$ that we are currently scheduling. For example, if we are studying a job trace, we can consider for $\mathcal{J}'$ the set of jobs that were scheduled during the previous week. If a job τ_i has category $C_i = 0$, it means that it is longer than all the jobs in the reference trace. Conversely, if τ_i has category $C_i = 1$, it means that it is shorter than all the jobs in the reference trace. Categories will allow us to select the target machine for the jobs. More precisely, we assign job τ_i to machine

$$M_i^c = \begin{cases} \lfloor C_i M_{use} \rfloor + 1 & \text{if } C_i < 1 \\ M_{use} & \text{otherwise} \end{cases} \tag{4.6}$$

- At any given time, we store S^+, the maximum stretch obtained so far. Recall that the stretch of a completed job τ_i is defined as the ratio $\frac{e_i - r_i}{w_i}$ of its response time $e_i - r_i$ (end time minus release time, or time spent in the system) over its length w_i. Maximum stretch is preferred to plain response time for fairness to short jobs.

We are now ready to describe the different decision processes for the events:

- **Job Arrival Event**: When a job arrives, we calculate its target machine M_i^c, and attempt to schedule it on this machine. If the job can start immediately, or is scheduled such that its estimated stretch does not exceed the maximum stretch S^+, we do schedule it on M_i^c. Otherwise, we choose the machine that can provides its

earliest start time among all machines whose index does not differ more than D^+ of that of M_i^c. In other words, we bound the distance from M_i^c to explore alternate target machines.

- **Job Completion Event**: When a job completes, no action is required since we have already scheduled the next jobs on each machine. However, since the number of cores of the machine changes, we need to update the total utilization of the machines and potentially change M_{alive} if necessary. We may also need to update S^+.
- **Machine Addition Event**: When a machine is added, no action is required unless M_{alive} corresponds to the previous number of machines and the total utilization is greater than 95%. In this case, we update M_{alive} and place all jobs that are in the waiting lists of machines and have not started into a global waiting list, and we re-allocate them: we process these jobs in ascending order of release dates r_i and allocate them to a machine using the same procedure as described above for job arrivals (which amounts to considering them as newly submitted jobs for mapping).
- **Machine Removal Event**: When a machine is removed and jobs are interrupted, we recalculate M_{use}, and reallocate all pending jobs as described in the previous point.

4.5.4 TargetASAP

In the previous section, we described TargetStretch, an algorithm that assigns a specific machine based upon the job length. This algorithm ensures that those machines that get killed only contain jobs among the shortest ones. However, some machines may be under-utilized if few jobs target them initially. Indeed, if the maximum stretch S^+ is very high, the algorithm will always favor the target machine, and thus could neglect a machine very close in terms of index that has idle cores, which is bad for goodput. For this reason, we have developed a second algorithm that differs from the previous one only in one aspect: when a job arrives, instead of scheduling it on its target machine M_i^c if its stretch is not higher than the maximum stretch, we proceed as follows:

- **Job Arrival Event:** If the job can start immediately upon arrival on its target machine M_i^c, we launch it there. Otherwise, we go through all machines whose index is at a distance smaller than D^+ and look for the closest machine to our target machine on which the job can run immediately, if one exists. If all machines around are full or do not have enough cores to run the job, we check whether the job can be scheduled on the target machine without increasing the maximum stretch S^+. If it is indeed possible, we schedule the job on the target machine. Otherwise, we schedule it on the machine that can start it at the earliest time among those within an acceptable distance.
- **Any Other Event:** Same as TargetStretch.

4.5.5 PackedTargetASAP

While TargetASAP solves the issue of under-utilized machines, it may leave some machines partially empty, in contrast to FirstFitAware that fills the machines perfectly by always using them in ascending order. For example, if jobs only use 60% of the available cores, FirstFitAware would use approximately 60% of the machines, whereas TargetASAP may use all the machines each at at 60% capacity. Furthermore, if a new job arrives and requires a large number of cores, TargetASAP may not be able to start it immediately while FirstFitAware could.

To address this issue, we group machines into packs. Instead of defining the target

machine as M_i^c as in Equation (4.6), we will round it to the nearest multiple of 5. The five machines corresponding to a pack will then be filled one after the other using D^+instead of all at once. For instance, if three jobs were assigned to machines 0, 1, and 2 by TargetASAP, all three jobs will be assigned to machine 0 by PackedTargetASAP, leaving machines 1 and 2 available for future jobs.

4.6 Experiments

We conduct experiments using an in-house simulator, both on synthetic traces and on actual workflow traces. An instance of the simulation consists of a combination of two traces, a resource variation trace that represents the number of machines alive at any given time (defined in Section 4.6.1), and a job trace as defined in Sections 4.6.2 and 4.6.3. We generate multiple simulation traces using the method given in Section 4.6.4. The experimental results are reported in Section 4.6.5. The code is publicly available at https://graal.ens-lyon.fr/~yrobert/experiments.zip.

4.6.1 Resource Traces

The generation of the resource variation trace takes three parameters: M_{avg}, M_{ra}, and ϕ. We start the resource variation trace at time 0 and end it at time T_{end}, where T_{end} is three weeks. The window begins at time T_{begin}after one week, so that the first week is a warmup. The number of machines follows a random walk that evolves periodically, with period ϕ, hence this number is a constant within each period and changes only at the end of a period.

Specifically, the average number of machines is M_{avg}. The total number of available machines always stays within the range $[M_{avg} - M_{ra}, M_{avg} + M_{ra}]$. It evolves randomly, staying constant, increasing or decreasing with equal probability unless one bound of the range is reached. In the latter case, the number of machines either stays constant or evolves in the unique possible direction, with same probability. Changes in the number of available machines always involve $step = \lfloor \frac{M_{ra}}{4} \rfloor$ machines, hence $step$ controls the magnitude of resource variation from one period to the next.

Formallly, let m_i the number of machines during the interval $[i\phi, (i+1)\phi]$ (period number i):

- $m_0 = M_{avg}$;
- If $m_i = M_{avg} - M_{ra}$, $\mathbb{P}\{m_{i+1} = m_i + step\} = \mathbb{P}\{m_{i+1} = m_i\} = \frac{1}{2}$;
- If $m_i = M_{avg} + M_{ra}$, $\mathbb{P}\{m_{i+1} = m_i - step\} = \mathbb{P}\{m_{i+1} = m_i\} = \frac{1}{2}$;
- Otherwise, $\mathbb{P}\{m_{i+1} = m_i - step\} = \mathbb{P}\{m_{i+1} = m_i\} = \mathbb{P}\{m_{i+1} = m_i + step\} = \frac{1}{3}$.

4.6.2 Job Traces: Borg

We experiment on traces of workloads running on Google compute cells that are managed by the cluster management software internally known as Borg [69]. We cut the Borg trace into slices of windows $[T_{begin}, T_{end}]$ of length 2 weeks. Inside each window, we keep the jobs τ_i for which $r_i \in [T_{begin} - w_i, T_{end}]$. We assume that the jobs τ_i that have been released before the beginning of the window ($r_i < T_{begin}$) have actually been running since their release date, hence job τ_i has $w_i - (T_{begin} - r_i)$ units of work remaining at time T_{begin}. Several jobs in the Borg trace are permanently running, we ignore them (or assume that these jobs execute on specific risk-less machines), and we focus on jobs that are lasting less than a day. Finally, we prune the traces so that the total work hours do not exceed the

maximum capacity of 26 machines, each with 24 cores, running during 2 weeks with full peak load. This is to match the dimensioning of the experimental window.

4.6.3 Job Traces: Synthetic Traces

A synthetic trace consists of a set of jobs $\mathcal{J}$ of n jobs, where each job τ_i is defined by the three parameters (r_i, c_i, w_i), where r_i is its release date, c_i its number of cores, and w_i its length.

Release dates r_i Similarly to the BORG trace, we study a two-week window. Because we want to start with a steady state, we add one week before the start of the window to generate jobs, therefore we assume the trace starts at time 0, and finishes at time T_{end} corresponding to 3 weeks, with T_{begin} corresponding to 1 week. For the beginning of the window at date T_{begin}, we will again assume that all jobs released at time $r_i < T_{begin}$ have been running for $T_{begin} - r_i$ units of time, for the jobs verifying $w_i > T_{begin} - r_i$.

We assume the jobs get released regularly, therefore if we generate a trace of n jobs for a total window $[0, T_{end}]$, one job will be released each T_{end}/n units of time. Therefore, $r_i = \frac{i}{n} T_{end}$.

Number of cores c_i The number of cores per jobs is drawn randomly, following roughly the distribution of the BORG traces, more specifically $\forall \tau_i \in \mathcal{J}, \mathbb{P}\{c_i = 1\} = \frac{1}{6}, \mathbb{P}\{c_i = 2\} = \frac{1}{3}, \mathbb{P}\{c_i = 4\} = \frac{1}{3}, \mathbb{P}\{c_i = 8\} = \frac{1}{6}$.

Length w_i Depending on the workflow type, the execution time of jobs is generated using different probability distributions. Similarly to BORG traces, the average job length T_{avg} is defined accordingly to n, so that the total number of machines required to process all jobs if there were no resource variations and if all machines were always used at maximum capacity is around 26 (e.g. such that the total core hour is around 209664, which means $T_{avg} = 209664/n$ hours). We create jobs along three different workflow types:

- For SYNTHETICUNIFORM, we generate the length of the jobs uniformly in $[0, 2T_{avg}]$, so that their average length is around T_{avg}.
- For SYNTHETICLOGSCALE, we draw the category c as a random integer in $[1, 4]$ such that $\mathbb{P}\{c = 1\} = 4\mathbb{P}\{c = 2\} = 4\mathbb{P}\{c = 3\} = 4\mathbb{P}\{c = 4\}$, then we draw the length uniformly in $[5^{c-1}K, 5^c K]$, where K is chosen so that the expected length of this random variable matches T_{avg}. Because c is not drawn uniformly, all categories have a non-negligible impact on the total core hours of the trace, although the category with the highest length is the most significant. We also experimented drawing c uniformly in $[1, 4]$ so that the longest jobs have a more significant impact. This version is called SYNTHETICLOGSCALEU; the results are very close to SYNTHETICUNIFORM; most of them are omitted in this thesis but available in [4].
- For SYNTHETIC3TYPES, we generate three types of jobs of three different length, t_{short}, t_{middle} and t_{high}, such that $t_{high} = 3t_{middle} = 9t_{short}$. We generate the jobs so that the total work hours of these three type of jobs is equal, i.e., $\mathbb{P}\{w_i = t_{short}\} = 3\mathbb{P}\{w_i = t_{middle}\} = 9\mathbb{P}\{w_i = t_{high}\}$. Therefore, we get $t_{short} = \frac{13}{27} T_{avg}$, $t_{middle} = \frac{39}{27} T_{avg}$, $t_{high} = \frac{117}{27} T_{avg}$, $\mathbb{P}\{w_i = t_{high}\} = \frac{1}{13}$, $\mathbb{P}\{w_i = t_{middle}\} = \frac{3}{13}$, and $\mathbb{P}\{w_i = t_{short}\} = \frac{9}{13}$.

4.6.4 Experimental Setup

An instance is defined by four parameters, that were defined in Sections 4.6.1 to 4.6.3:

- The average number of machines $M_{avg} = 24$, to be slightly below the number of machines required in average.
- The period of machine variation $\phi = 1200s$: one change every 20 minutes.
- The range of machine variation $M_{ra} = 8$, therefore the machines will always be in $[M^- = 16, M^+ = 32]$; half the machines are safe.
- The number of cores per machine $N = 24$.

One last parameter for the synthetic traces is the number of jobs $n = 20000$. The values above are the ones used by default, and we further study the impact of each parameter separately. Specifically, we consider:

- $M_{avg} \in [20, 22, 24, 26, 28]$ (with $\phi = 1200s$, $M_{ra} = 8$ and $N = 24$). This experiment is to explore what happens if the number of machines is too small to process the workload, or, in contrary if there are (in average) enough machines to process every job.
- $\phi \in [400, 1200, 3600, 10800, 32400]$ (between 7 minutes and 9 hours, with $M_{avg} = 24$, $M_{ra} = 8$ and $N = 24$). The average number of available machines remains the same, but the number of job interruptions is likely to decrease when ϕ increases.
- $M_{ra} \in [4, 6, 8, 12, 16]$ (with $M_{avg} = 24$, $\phi = 1200s$, and $N = 24$): in the most extreme scenarios, only 8 machines are safe while we could have up to 40 machines available.
- $N \in [8, 16, 24, 32, 48]$ (with $\phi = 1200s$ and $M_{ra} = 8$). Here we do not keep the same number of machines when we vary the number of cores per machine, because we have already studied the impact of the total number of resources when varying M_{avg}. Instead, we scale the range of $M_{avg} \in [72, 36, 24, 18, 12]$ and of $M_{ra} \in [24, 12, 8, 6, 4]$ accordingly, so that $M_{ra} \times N$ remains constant and $M_{ra} = \frac{M_{avg}}{3}$ (i.e., half of the machines are safe). This experiment assesses the impact of the target platform configuration, comparing few machines with many cores against many machines with fewer cores.
- $n \in [8000, 14000, 28000, 40000]$ (with $M_{avg} = 24$, $\phi = 1200s$, $M_{ra} = 8$ and $N = 24$). This experiment explores the impact of system load.

For each set of parameters, we run each heuristic under six 2-week traces per workflow type, each of them with 30 different variation traces. Results are shown using boxplots where the average is represented by a star, the boxes show the 25th to 75th percentiles, and the whiskers indicate the 10th and 90th percentiles.

Finally, while GOODPUT remains the major focus, we also report the performance of each heuristic for three other interesting metrics:

- MAXIMUMSTRETCH: Recall from Section 4.5.3 that the stretch of a completed job $\tau_i \in \mathcal{J}_{comp,T}$ is defined as the ratio $\frac{e_i - r_i}{w_i}$ of its response time $e_i - r_i$ (end time minus release time, or time spent in the system) over its length w_i. The maximum stretch corresponds to S^+defined earlier taken at time T_{end}, e.g., $S^+ = \max_{\tau_i \in \mathcal{J}_{comp,T_{end}}} \left(\frac{e_i - r_i}{w_i} \right)$.
- ABORTEDVOLUME is defined as the total amount of core hours lost because of job interruptions, normalized by the total amount of core hours that were available. More precisely, if we define a family of events corresponding to all interruptions $\mathcal{I}$, where each interruption $i_k \in \mathcal{I}$ corresponds to a time t_k and a related job τ_k that started at time s_k with c_k cores, then $\text{ABORTEDVOLUME} = \frac{\sum_{i_k \in \mathcal{I}} (t_k - s_k) c_k}{N \sum_{t \in [0, T-1]} M_{alive}(t)}$. Note that we always have

  $$\text{GOODPUT} + \text{ABORTEDVOLUME} \leq 1,$$

  because we have normalized by to the total core hours of work available during the

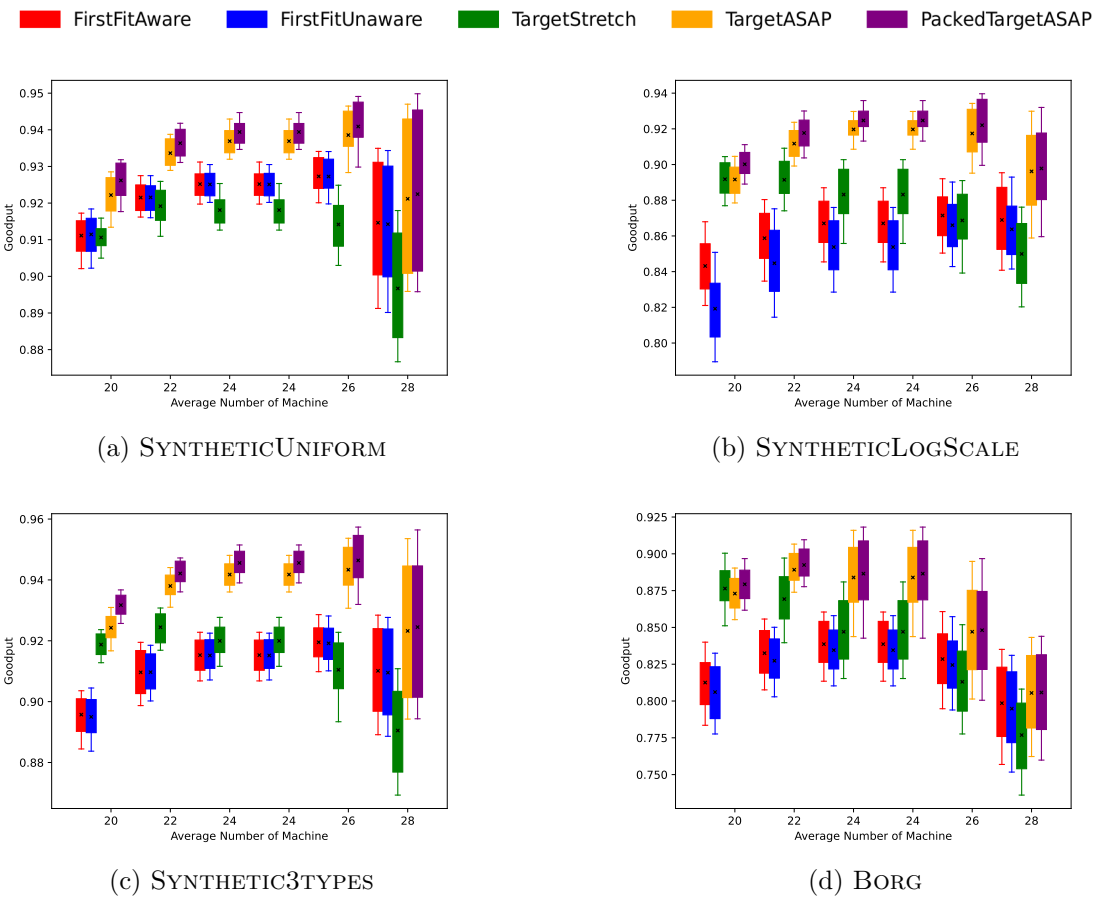
processing. This sum may be lower than 1 because cores are sometimes idle.

- AVERAGEABORTEDTIME is the average time lost at each interruption, defined as $\frac{\sum_{i_k \in \mathcal{I}} (t_k - s_k)}{card(\mathcal{I})}$.

Two more metrics are omitted in this thesis but can be found in [4].

4.6.5 Experimental Results

In this section, we provide the experimental results that follow the setup described above.



(a) SYNTHETICUNIFORM (b) SYNTHETICLOGSCALE

(c) SYNTHETIC3TYPES (d) BORG

Figure 4.2: GOODPUT when varying the number of machines M_{avg}.

Varying the number of machines M_{avg} Figure 4.2 shows the results of all five heuristics for the GOODPUT metric when the average number of machines M_{avg} vary. First, note that GOODPUT corresponds to the total work successfully executed divided by the number of available core hours. Therefore, we compute the proportion of machines used over time, rather than the total amount of work done. This explains why the results of all the heuristics decrease between 26 and 28 average machines in terms of goodput. For $M_{avg} = 28$, even if all jobs were completed (which is almost the case for TARGETASAP and PACKEDTARGETASAP), the goodput would not be high because not all machines are fully utilized. The increasing portion between 20 and 26 machines can be explained by the fact that M_{ra} is fixed. Thus, the proportion of machines experiencing variability is higher for a low number of average machines, resulting in more aborted work.

Regarding the differences between the heuristics, we observe that the TARGETASAP

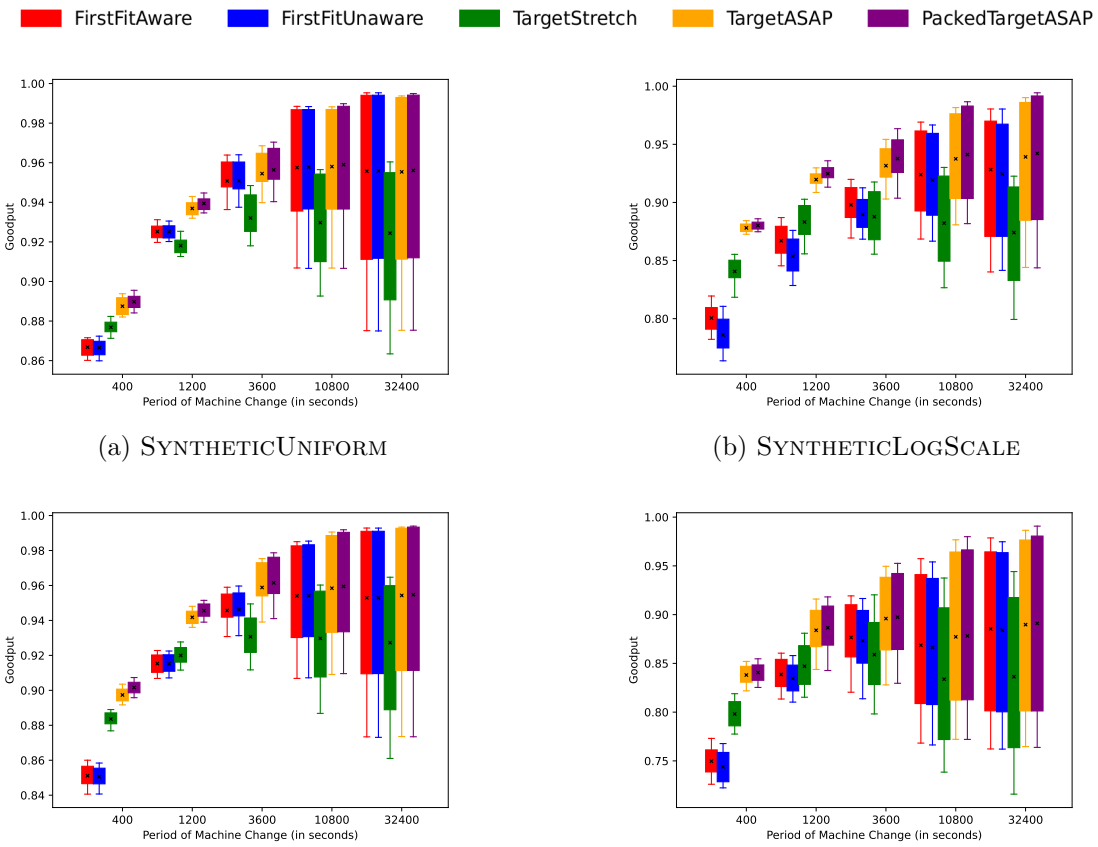
and PackedTargetASAP heuristics are always better than the two competitors, FirstFitAware and FirstFitUnaware. In fact, this observation is over all the experimental results, including those in the appendices. The overall trend of TargetASAP, PackedTargetASAP, FirstFitAware, and FirstFitUnaware is similar because these four heuristics share the characteristic of having a good allocation of jobs on the machines, with cores rarely being idle when jobs are waiting. The difference in results is due to the fact that FirstFitAware and FirstFitUnaware do not specifically preserve the longest jobs, which leads to more interrupted work. TargetStretch has a different behavior because even though it takes into account the length of jobs that need to be preserved, it lacks flexibility, and some machines tend to remain partially inactive even when jobs are waiting. This phenomenon increases as the number of machines grows, because the likelihood of having discrepancy between machines increases with the number of jobs.

Finally, TargetASAP and PackedTargetASAP have fairly similar results, with a slight advantage for PackedTargetASAP. Similarly, FirstFitAware and FirstFitUnaware have fairly similar results (except for SyntheticLogScale where FirstFitUnaware is much lower). While this may seem surprising at first, it can be explained by the fact that when there are too many jobs for the number of machines, they are all either turned off or saturated; then, not knowing which machine will be turned off first is not penalizing. In the opposite case, there are enough machines available to support interruptions and re-executions. The workflow type generally has a limited impact on the relative performance of the heuristics, both for synthetic traces or for Borg. There are some minor relative performance details, e.g. TargetStretch is at the same level or even better than TargetASAP for SyntheticLogScale and Borg, while it is below FirstFitAware and FirstFitUnaware for SyntheticUniform. Altogether, the differences in goodput between the heuristics differ from a workflow type to another, from around 2% for SyntheticUniform and Synthetic3types and up to almost 10% for Borg, showing that the potential gain is substantial. The heuristics preserving long jobs should improve the Goodput in many practical scenarios.

Varying the period ϕ of machine variation Figure 4.3 shows the results of all five heuristics for the Goodput metric when the period ϕ of machine variation varies. Once again, we observe in this figure that the impact of the workflow type seems limited since the four figures are generally similar, except for the scale of the y-axis. For FirstFitAware and FirstFitUnaware, the goodput can reach values lower than 75% for SyntheticUniform, while it remains above 84% for Synthetic3types and Borg.

We note that the goodput generally increases with the period ϕ. This is logical since the less machine changes, the lower the risk of interruption, even if the average number of machines remains unchanged. We also observe a strong increase in variability for a given heuristic and set of parameters when the period is high. This is due to the fact that the goodput is scaled by the total available cores. The longer the period, the fewer states the random walk of the number of machines will take, and therefore the average of this specific random walk will be further away from its statistical average. In other words, there is a high variability in total available core hours: if the random walk stays with high values of machine numbers, all jobs can be executed; but since the system load is fixed, the goodput will be reduced.

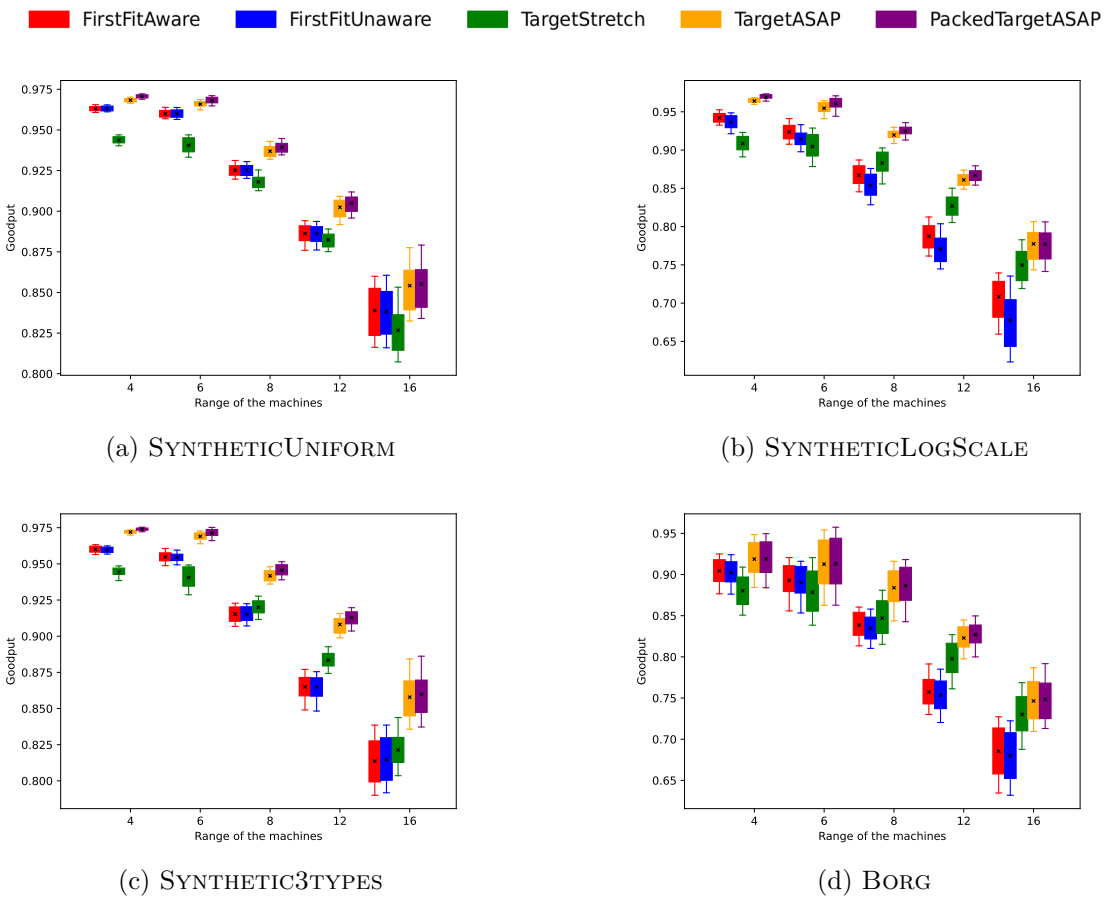
Finally, we note that the relative performance of TargetStretch improves when the period is low. Indeed, the lower the period, the more interesting it is to be safe and preserve long jobs. Conversely, the higher the period, the more problematic its shortcomings on the overall quality of the schedule.



(a) SyntheticUniform (b) SyntheticLogScale

(c) Synthetic3types (d) Borg

Figure 4.3: Goodput when varying the period ϕ of machine variation.



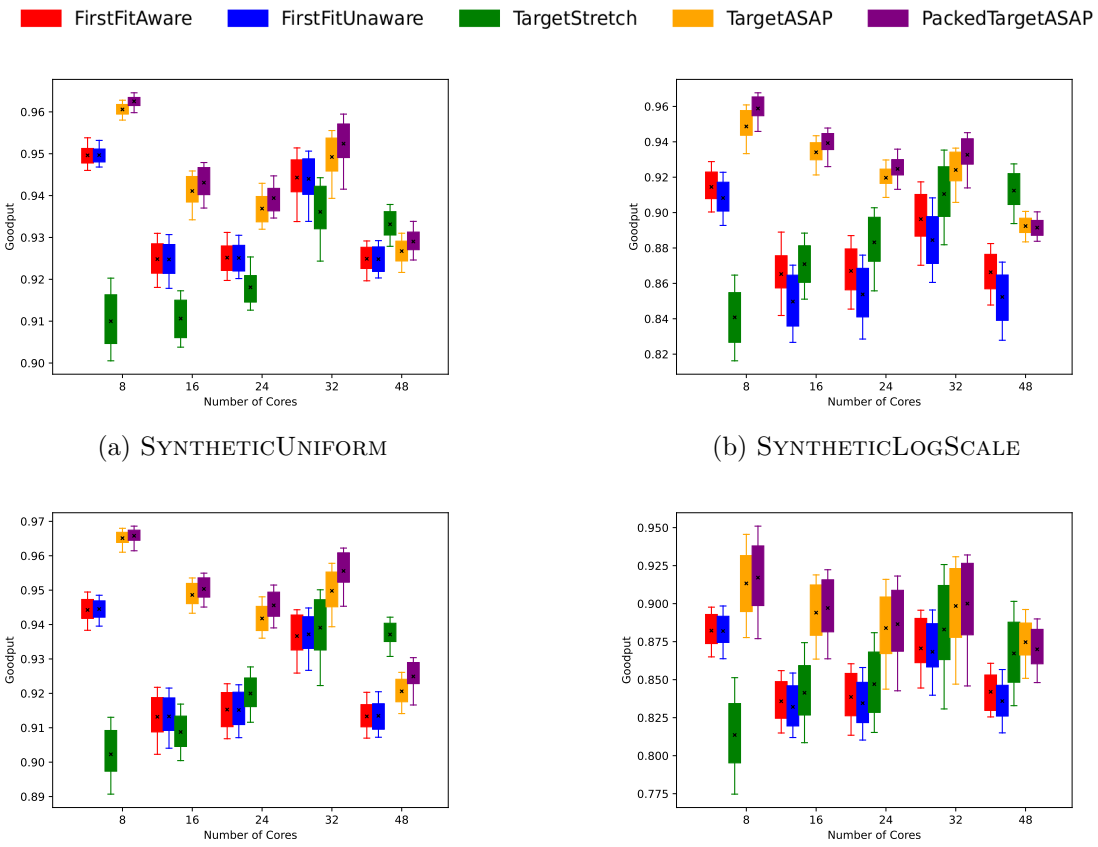
(a) SyntheticUniform (b) SyntheticLogScale

(c) Synthetic3types (d) Borg

Figure 4.4: Goodput when varying the range M_{ra} of machine variation.

Varying the range M_{ra} of machine variation Figure 4.4 shows the results of all five heuristics for the Goodput metric when the range of machine variations M_{ra} vary. Of course, the higher the machine range M_{ra}, the lower the goodput, since more jobs are interrupted. We observe this for all types of workflows and for all heuristics, which generally maintain their relative performance. We simply note that TargetStretch is slightly less impacted by the increase in range, again because it is scheduling more safely than the other heuristics.

Varying the number of cores per machine N Figure 4.5 shows the results of all five heuristics for the Goodput metric when the number of cores per machines N varies. In this experiment, we varied the number of cores per machine while keeping the average total number of cores constant. This means that the number of machines is scaled inversely proportional to the number of cores. While it is difficult to extrapolate a general dynamic related to the number of cores for the FirstFitAware, FirstFitUnaware, TargetASAP, and PackedTargetASAP heuristics, this experiment confirms the quality of TargetStretch when the number of cores is high (and therefore, when the number of machines is low.) This is because the drawback of this heuristic, that sometimes leads to partially unused machines even when jobs are waiting, is drastically reduced with the number of machines is low; TargetStretch becomes the best heuristic for all workflow types except Borg (where it is close) for a high number of cores and therefore a low number of machines.



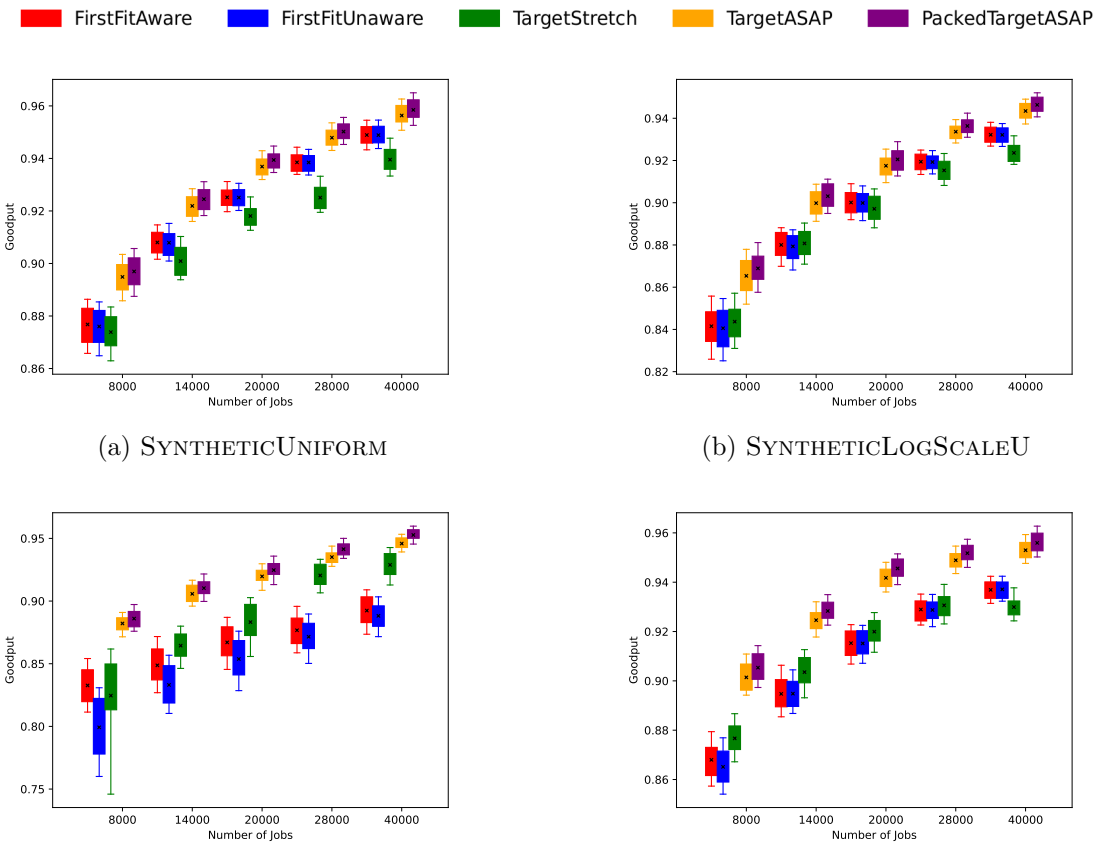
(a) SyntheticUniform

(b) SyntheticLogScale

(c) Synthetic3types

(d) Borg

Figure 4.5: Goodput when varying the number of cores per machines N.

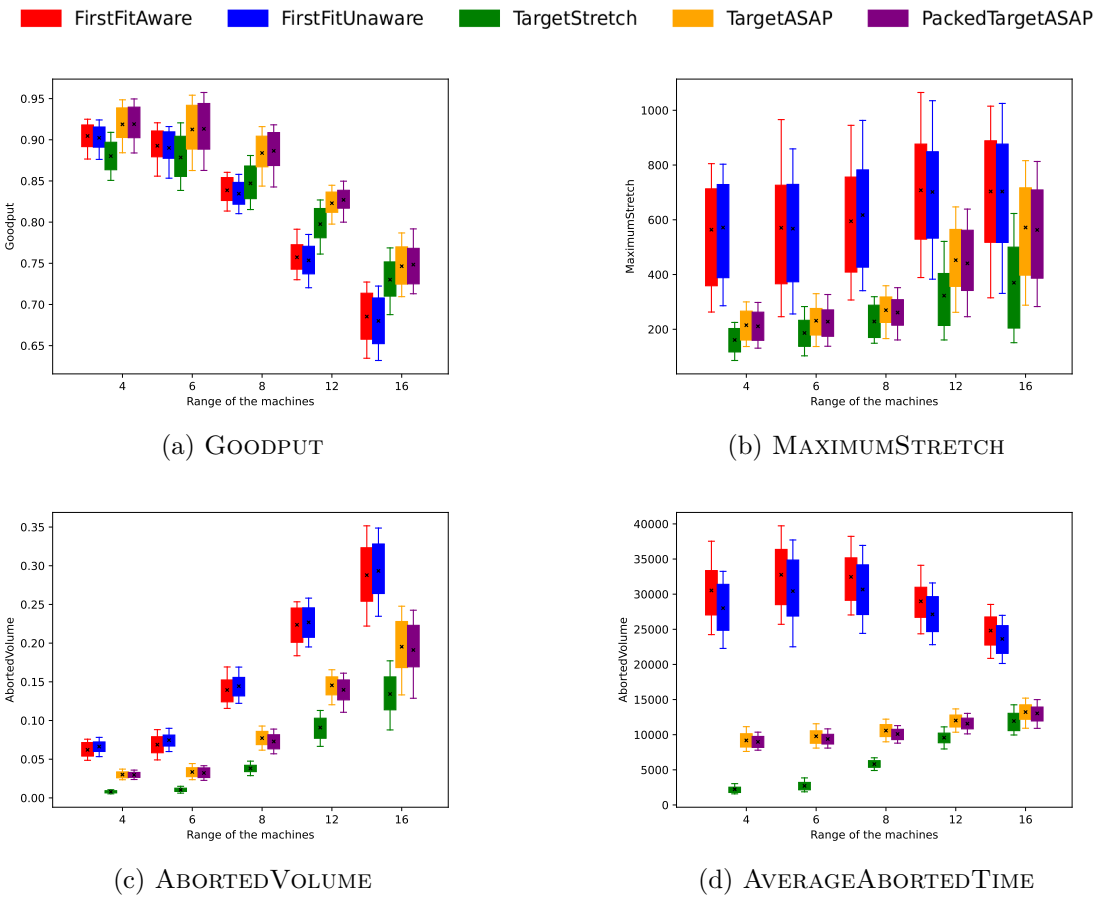


(a) SyntheticUniform (b) SyntheticLogScaleU

(c) SyntheticLogScale (d) Synthetic3Types

Figure 4.6: Goodput when varying the number n of synthetic jobs.

Varying the number n of synthetic jobs Figure 4.6 shows the results of all five heuristics for the GOODPUT metric when the number of synthetic jobs n varies. The goodput increases when the number of jobs increases. This is because the total volume of work is constant, and the more jobs, the shorter they are. Thus, the work lost during a job interruption is on average lower regardless of the heuristic. The relative performance between heuristics is the same for all types of workflows and all numbers of jobs, with FIRSTFITAWARE and FIRSTFITUNAWARE being worse than TARGETSTRETCH, which is itself worse than TARGETASAP and PACKEDTARGETASAP. Once again, the advantage of the packed version PACKEDTARGETASAP is visible, although relatively small compared to TARGETASAP. This experiment does not apply to BORG since the jobs are fixed in this trace, so we decided to replace BORG with SYNTHETICLOGSCALEU, which is the last synthetic workflow type we studied. In general, the results of SYNTHETICLOGSCALEU are particularly close to SYNTHETICUNIFORM (more details can be found in [4]).



(a) GOODPUT (b) MAXIMUMSTRETCH

(c) ABORTEDVOLUME (d) AVERAGEABORTEDTIME

Figure 4.7: Different metrics to analyze the results on BORG for varying range of machine variation M_{ra}.

Exploring other metrics Figure 4.7 shows the results of all five heuristics for BORG when the range M_{ra} of machine variation varies, and for different metrics: GOODPUT (reproduced from Figure 4.4 for convenience), MAXIMUMSTRETCH, ABORTEDVOLUME, and AVERAGEABORTEDTIME. First and as expected, the results for all metrics are degraded by increasing the range M_{ra} for each of the five heuristics. While TARGETSTRETCH is clearly worse than TARGETASAP and PACKEDTARGETASAP for the goodput, it turns out to be the best heuristic for MAXIMUMSTRETCH. Indeed, it is the heuristic that best preserves

long jobs, which is reflected both in the total volume aborted AbortedVolume and in the average time lost per interrupted job AverageAbortedTime, where TargetStretch achieves the lowest values. Obviously, FirstFitAware and FirstFitUnaware, which have no consideration for job length, perform poorly for these metrics.

More details for these metrics are omitted in this thesis but can be found in [4]. Overall, TargetStretch is not always better than TargetASAP and PackedTargetASAP for the maximum stretch: this depends on the experiments performed. On the contrary, the basic heuristics FirstFitAware and FirstFitUnaware are always the ones that perform the worst for all other metrics. We observe that the difference in maximum stretch is not at all negligible, with a factor of 2 to 3 compared to TargetASAP, PackedTargetASAP, and TargetStretch.

It is also worth studying the sum of Goodput and AbortedVolume. For FirstFitAware, FirstFitUnaware, TargetASAP, and PackedTargetASAP, this sum is close to 1 generally, which means that the machines are often used to their maximum capacity. This is not the case for TargetStretch, which, for example, is well below TargetASAP and PackedTargetASAP for both metrics, which means that there is a frequent under-utilization of machines, even though it better preserves long jobs.

4.6.6 Summary

In summary, there are scenarios for which TargetStretch can be a decent heuristic, for instance when the number of machines is low or the variability is high. In this case, it is necessary to be very careful in sorting jobs by length in order to preserve the longest ones and not loose too much work due to interruptions. However, the lack of flexibility of TargetStretch can be problematic when the impact of machine variation is not critical, which is the case for most of the experiments we have done. Then, although it was designed to preserve the maximum stretch MaximumStretch, its performance is generally comparable to TargetASAP and PackedTargetASAP for this metric.

Furthermore, TargetASAP and more particularly PackedTargetASAP (which is slightly better) offer a greater flexibility. Hence, even if they are slightly less precise for preserving the longest jobs, they allow for an excellent overall utilization of machines and a good preservation of the longest jobs, which makes them the best heuristics in almost all scenarios. They are better than their standard competitors FirstFitAware and FirstFitUnaware, in all cases. The superiority of TargetASAP and PackedTargetASAP is significant: up to 10% increase in goodput and a maximum stretch two to three times smaller. We conclude that the use of FirstFitAware and FirstFitUnaware should be reconsidered with variable resources.

4.7 Summary and Future Work

With growing variation in power cost, availability, and carbon-intensity, driven by integration of more renewable generation to the power grid, the incentives to operate datacenters as variable loads, producing variable computing capacity are growing. Resource management (schedulers) must be advanced to handle large-scale and perhaps increasingly frequent capacity variation, yet achieve high utilization of the available capacity. The primary challenge is that when capacity decreases, running jobs may need to be terminated to meet the required power load reduction.

We present online risk-aware scheduling strategies to preserve performance in this variable capacity environment. Specifically, we design novel risk-aware scheduling and

mapping algorithms that assign the right job to the right machine, optimizing for the system's goodput. Our algorithms employ a variety of techniques to mitigate the impact of resource variation, including maintaining a risk index per machine, mapping longer jobs to safer machines, maintaining local queues at machines, re-executing interrupted jobs on new machines, and redistributing pending jobs as resource capacity increases. Our assessment using workload trace from Google's Borg system and three synthetic traces shows significant gains over first-fit algorithms with up to 10% increase in goodput, with no loss in complementary metrics, such as the maximum and average stretch. We conclude that standard first-fit algorithms are insufficient for future variable capacity environments and require re-design in order to maintain the expected level of performance.

Directions for future work are bountiful; they include exploration of different workloads, different job execution models (e.g. migration or deferral), different variation models, and the recent interesting direction of malleable workloads.

Part II

Revisiting checkpoint and I/O bandwidth sharing strategies

Chapter 5

Checkpointing strategies to tolerate non-memoryless failures on HPC platforms

In the first part of this thesis (Chapters 2 to 4), we delved into resilience without checkpointing, where tasks or jobs must be redone completely when they failed or needed to be cut, emphasizing mostly on task scheduling, error detection, and recovery. From this chapter and onward, we will consider checkpointing strategies for parallel applications subject to failures to avoid re-executing an application from scratch. This chapter focuses on tightly-coupled parallel applications that are preemptible, meaning that one can take a checkpoint at any instant. The optimal strategy to minimize total execution time, or makespan, is well known when failure inter-arrival times obey an Exponential distribution, but it is unknown for non-memoryless failure distributions. We explain why the latter fact is misunderstood in recent literature. We propose a general strategy that maximizes the expected efficiency until the next failure, and we show that this strategy achieves an asymptotically optimal makespan, thereby establishing the first optimality result for arbitrary failure distributions. Through extensive simulations, we show that the new strategy is always at least as good as the Young/Daly strategy for various failure distributions. For distributions with a high infant mortality (such as LogNormal with shape parameter $k = 2.51$ or Weibull with shape parameter 0.5), the execution time is divided by a factor 1.9 on average, and up to a factor 4.2 for recently deployed platforms. This chapter corresponds to Submission [S1] (See Chapter 9).

5.1 Introduction

Checkpoint/restart is the standard technique to protect applications running on High Performance Computing (HPC) platforms. Such platforms experience several failures[1] per day [36, 61, 139, 142] per day. After each failure, the application executing on the faulty processor (and likely on many other processors for a large parallel application) is interrupted and must be restarted. Without checkpointing, all the work executed for the application is lost. With checkpointing, the execution can resume from the last checkpoint, after some downtime (enroll a spare to replace the faulty processor) and a recovery (read the checkpoint).

Consider a parallel application executing on a platform whose processors are subject to failures. How frequently should it be checkpointed so that expected total execution time, or makespan, is minimized? There is a well-known trade-off: taking too many checkpoints leads to a high overhead, especially when there are few failures, while taking too few

[1] Failures are also called *fail-stop errors*

checkpoints leads to a large re-execution time after each failure. However, the optimal strategy to minimize the expected makespan is known only when failure inter-arrival times, or IATs for short[2], obey an Exponential distribution on each processor. In that case, the optimal checkpointing period is known and can be expressed with a complicated formula (see Sections 5.3.1 to 5.3.3). In practice, the optimal checkpointing period is approximated by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$ [47, 173], where μ is the application Mean Time Between Failures (MTBF) and C is the checkpoint duration. The Young/Daly formula is widely used across a variety of applications and platforms (see [22] for a survey) and represents a major progress compared to naive strategies where each application would checkpoint, say, every hour, independently of the values of its MTBF μ and checkpoint duration C.

This chapter revisits checkpointing strategies for parallel applications on platforms subject to failures that obey arbitrary probability distributions. This is a very important topic because the most accurate probability distributions to model processor failures are LogNormal [79] and Weibull [138, 139, 150, 151] instead of Exponential. For instance, failure traces from Los Alamos National Laboratory are best fit by Weibull distributions of different shapes [57]. However, dealing with non-memoryless distributions induces dramatic difficulties when aiming at optimality. Indeed, if each processor experiences failures distributed according to some non-memoryless distribution, then the platform as a whole will NOT experience failures distributed according to the same (scaled) distribution. This is because after each failure, only the processor struck by that failure is replaced by a fresh (spare) processor. The other thousands of processors in the platform are not replaced and continue execution; even if we wanted to replace them all by fresh processors, we would not have enough spares for such a massive replacement. Hence, after a few failures have struck the platform, processors have a different history, meaning that the time since the last failure is different from one processor to another. As a consequence, the platform experiences failures which do no longer obey the same probability distribution from one failure to the next. Worse, computing the probability distribution of the time at which the next platform failure will strike would require a massive convolution over all processors. In practice, such a computation is out-of-reach as soon as the number of processors exceeds a few dozens.

The striking difference between the Exponential distribution and any other non-

[2]IATs are the times elapsed between two consecutive failure events (or until the first failure at the start of the application).

$\mathcal{D}$	Probability distribution of failures
T_{plat}	Age of the platform
T_{base}	Base time of the application (without failures nor checkpoints)
C	Checkpoint time
R	Recovery time
D	Downtime
$\vec{\tau}$	History vector (or age) of all processors
N	Number of checkpoints/segments currently planned.
W	Work of a segment
$\mathbb{P}_{suc}(x\|\vec{\tau})$	Probability that there are no failures in the next x units of time, $\mathbb{P}_{suc}(x\|\vec{\tau}) = \prod_{i=1}^{p} \mathbb{P}(X \geq x + \tau_i \| X \geq \tau_i)$
$\vec{\tau^*}, C^*, W^*, \cdots,$	Variables expressed in numbers of quanta (integers)

Table I: Summary of main notations for Chapter 5.

memoryless distribution is further detailed in Sections 5.3.5 and 5.3.6. In a nutshell, when each processor experiences failures obeying an Exponential distribution, so does each parallel application (and the whole platform). Take a parallel application with p processors; if processor IATs are $\text{Exp}(\lambda)$, then application IATs are $\text{Exp}(p\lambda)$; the MTBF for each individual processor is the expectation of $\text{Exp}(\lambda)$, namely $\mu_{ind} = \frac{1}{\lambda}$, while the application MTBF is $\mu = \frac{\mu_{ind}}{p}$. The knowledge of the distribution of application IATs is key to derive an analytic expression for the optimal checkpointing period, and to show that the Young/Daly formula is an accurate first-order approximation. But when each processor experiences failures obeying any other distribution, then platform failures are no longer identically distributed, and not much is known about the optimal checkpoint strategy. This does not prevent to use the Young/Daly formula, because it relies on the application MTBF only, not on any probability distribution. The application MTBF can still be computed as $\mu = \frac{\mu_{ind}}{p}$, as shown in Section 5.3.6. What is completely unknown though is the accuracy of the Young/Daly formula for a non-memoryless distribution.

To the best of our knowledge, this chapter is the first to provide a provenly correct strategy for arbitrary distributions. The main contributions are the following:

- A synthetic overview of known results for Exponential distributions, some of which being frequently rediscovered;
- A detailed explanation of why non-memoryless distributions require a fully different approach;
- The design of a new checkpointing strategy, NextStep, which is asymptotically optimal for arbitrary distributions;
- A practical and fast implementation of NextStep through time discretization and numerical approximation;
- A detailed experimental comparison with the standard Young/Daly approach.

This work focuses on the classical coordinated checkpointing protocol, which has well-known limitations because of the potential bottleneck incurred when all processors transfer data to stable storage simultaneously [35,36]. Section 5.2.3 surveys more complicated (multi-level) approaches designed for large-scale platforms. Future work will aim at extending the NextStep strategy to such approaches.

The chapter is organized as follows. We first survey related work in Section 5.2. Then, we provide background on checkpointing parallel applications with Exponential or non-memoryless distributions in Section 5.3. We detail the design of the checkpointing strategy NextStep in Section 5.4, and show that it is asymptotically optimal for arbitrary distributions. The experimental evaluation in Section 5.6 presents extensive simulation results comparing NextStep and the usual approach à la Young/Daly. Finally, we conclude in Section 5.7.

5.2 Related work

We survey related work related to checkpointing preemptible applications in Section 5.2.1 and task-based applications in Section 5.2.2. Section 5.2.3 is devoted to the presentation of several extensions of the standard approach.

5.2.1 Checkpointing preemptible parallel applications

Checkpoint-restart is one of the most widely used strategy to deal with failures. Several variants of this policy have been studied; see [80] for an overview. The natural strategy is to checkpoint periodically, and one must decide how often to checkpoint, i.e., derive the

optimal checkpointing period. An optimal strategy is defined as a strategy that minimizes the expectation of the execution time of the application. For a preemptible application, where one can checkpoint at any time, the classical formula due to Young [173] and Daly [47] states that the optimal checkpointing period is $W_{YD} = \sqrt{2\mu C}$, where μ is the application MTBF and C the checkpoint cost. This formula is a first-order approximation. For memoryless failures, Daly provides a second-order, more accurate, approximation in [47], while our previous work [31] provides the optimal value; both [47] and [31] use the Lambert function, whose Taylor expansion is key to assess the accuracy of the Young/Daly formula. The derivation in [31] is based on Equation (26) (see Section 5.3.2), a formula rediscovered ten years later, with a quite different proof based on a Markov model, in [142].

As explained in Section 5.3.6, non-memoryless failures are more difficult to deal with for parallel applications. Several papers study non-periodic checkpointing strategies, either with a single processor or with total rejuvenation of all processors after a failure [82,113,129]. A recent paper [111] also uses full rejuvenation while [150] wrongly assumes independent and identically distributed (IID) failures for a range of classic distributions, including Weibull and LogNormal, which are not memoryless (see Section 5.3.6). An unorthodox approach is used in [64], where it is assumed that the failures striking the whole platform obey a Weibull distribution; this is misleading for two reasons: (i) it is not clear what is the failure distribution on each individual processor; and (ii) after one processor is struck by a failure and rejuvenated, the platform failure distribution does not remain Weibull (see a more detailed discussion in Section 5.3.6).

In order to deal with non-memoryless failures, the NextFailure problem is studied in [31], where the goal is to maximize the expected amount of work completed before the next failure. This problem is solved using a dynamic programming algorithm, and it is used as a solution to the initial problem of makespan minimization. Simulations are done with Exponential and Weibull distributions, showing that the proposed algorithm outperforms existing solutions with Weibull distributions. In this chapter, we propose to maximize the expected efficiency rather than the expected work, with our new NextStep heuristic. This requires a much more subtle approach, but is key to proving asymptotic optimality.

5.2.2 Checkpointing task-based applications

Going beyond preemptible applications, some works have studied task-based applications, using a model where checkpointing is only possible right after the completion of a task. The problem is then to determine which tasks should be checkpointed. This problem has been solved for linear workflows (where the task graph is a simple linear chain) by Toueg and Babaoglu [158], using a dynamic programming algorithm. This algorithm was later extended in [21] to cope with both fail-stop and silent errors simultaneously. Another special case is that of a workflow whose dependence graph is arbitrary but whose tasks are parallel tasks that each executes on the whole platform. In other words, the tasks have to be serialized. The problem of ordering the tasks and placing checkpoints is proven NP-complete for simple join graphs in [7], which also introduces several heuristics. For general workflows, deciding which tasks to checkpoint has been shown #P-complete [76], but several heuristics are proposed in [77].

5.2.3 Extensions: multi-criteria, hierarchical checkpointing, independence

For completeness, in this section we briefly reference several works that go beyond makespan optimization and independent failures. First, other optimization criteria have been considered in the literature. Indeed, I/O is a scarce resource on modern platforms, and

several works aim at minimizing I/O volume while enforcing an efficient checkpoint for makespan [81, 99]. Similarly, energy-makespan bi-criteria optimization has been addressed in [57, 67].

Next, to reduce I/O overhead, various two-level checkpointing protocols have been studied [51, 143]. Some authors have also generalized two-level checkpointing to account for an arbitrary number of levels [15, 20, 49, 122].

As for failure independence, the standard model assumes IID failure IATs, on each processor, with a common distribution $\mathcal{D}$. While it is reasonable to assume that IATs are identically distributed on a given processor, because the faulty processor is rejuvenated (replaced by a spare) after each failure, it is very questionable to assume that IATs are independent across the platform. As for *temporal* dependence, it has been observed many times that when a failure occurs, it may trigger other failures that will strike different system components [14, 79, 156]. As an example, a failing cooling system may cause a series of successive crashes of different processors. Also, an outstanding error in the file system will likely be followed by several others [105, 140]. As for *spatial* dependence, it is clear that the overheating of some processor in a cabinet is quite likely to be followed by the overheating of neighbor processors (which comes atop of a temporal dependence as well!) Bautista-Gomez et al. [14] have studied nine systems, and they report periods of high failure density in all of them. They call these periods *cascade failures*. This observation has led them to revisit the temporal failure independence assumption, and to design bi-periodic checkpointing algorithms that use different periods in normal (failure-free) and degraded (with failure cascades) modes. [156] introduces a dynamic strategy called *lazy checkpointing* to adjust to changes in the failure rate. Another approach has been proposed in [10], using quantiles of consecutive IAT pairs.

5.3 Background

This section overviews known results for checkpoint strategies. We cover uni-processor and multi-processor applications, either with Exponential failure distributions or with arbitrary failure distributions. Beforehand, we detail the platform and application model.

5.3.1 Model

Platform and applications

We consider a large parallel platform with P identical processors. We point out that this work is agnostic to the granularity of the computing resources, which can vary from individual cores up to complete multicore nodes equipped with several GPUs. Each of the P processors is subject to failures. A failure interrupts the execution of the processor and provokes the loss of its whole memory. We consider parallel applications that can be checkpointed at any time. In scheduling terminology, the applications are preemptible. Consider a parallel application running on several processors: when one of these processors is struck by a failure, the state of the application is lost, and execution must restart from scratch, unless a fault-tolerance mechanism has been deployed. The classical technique to deal with failures makes use of a checkpoint-restart mechanism: the state of the application is periodically checkpointed, i.e., all participating processors take a checkpoint simultaneously. This is the standard coordinated checkpointing protocol, which is routinely used on large-scale platforms [39], where each processor writes its share of application data to stable storage (checkpoint of duration C). When a failure occurs, the platform is unavailable during a downtime D, which is the time to enroll a spare processor that will

replace the faulty processor [47, 80]. Then, all application processors (including the spare) recover from the last valid checkpoint in a coordinated manner, reading the checkpoint file from stable storage (recovery of duration R). Finally, the execution is resumed from that point on, rather than starting again from scratch. Note that failures can strike during checkpoint and recovery, but not during downtime (otherwise we can include the downtime in the recovery time). When a failure hits a processor, that processor is replaced by a spare. This amounts to start anew with a fresh processor. In the terminology of stochastic processes, the faulty processor is rejuvenated. However, all the other processors are not rejuvenated: this would be infeasible due to the multitudinous spares needed!

Failures

We assume that each processor experiences failures whose IATs follow IID random variables obeying an arbitrary probability distribution $\mathcal{D}$. We only assume that $\mathcal{D}$ is continuous and of finite expectation and variance, a condition satisfied by all standard distributions. We let μ_{ind} denote the expectation of $\mathcal{D}$, also known as the individual processor MTBF. Even if each processor has an MTBF of several years, large-scale parallel platforms are composed of so many processors that they will experience several failures per day [36, 61]. Hence, a parallel application using a significant fraction of the platform will typically experience a failure every few hours.

Checkpointing strategy

Given a parallel application with base time T_{base} without checkpoints nor failures, the optimization problem is to decide when and how often to checkpoint in order to minimize the expected execution time of the application. The application base time is divided into N segments of length W_i, $1 \leq i \leq N$, each followed by a checkpoint of length C. Of course $\sum_{i=1}^{N} W_i = T_{base}$. Throughout the chapter, we add a final checkpoint at the end of the last segment, e.g., to write final outputs to stable storage. Symmetrically, we add an initial recovery when re-executing the first segment of an application (e.g., to read inputs from stable storage) if it has been struck by a failure before completing the checkpoint. Adding a last checkpoint and an initial recovery brings symmetry and simplifies formulas, but it is not at all mandatory: see Section 5.3.4 for an extension relaxing either or both assumptions.

5.3.2 Uni-processor application and Exponential failure distribution

This is the simplest case. Consider an application $\mathcal{A}$ executing on a single processor experiencing failures whose IATs follow an Exponential distribution $\mathcal{D} = \text{Exp}(\lambda)$ of parameter $\lambda > 0$, whose probability density function (PDF) is $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. The processor MTBF is $\mu_{ind} = \frac{1}{\lambda}$. The optimal checkpointing strategy, i.e., the strategy minimizing the expected execution time, can be derived as shown below.

Lemma 26. *The expected time $\mathbb{E}(W, C, R)$ to execute a segment of W seconds of work followed by a checkpoint of C seconds and with recovery cost R seconds is*

$$\mathbb{E}(W, C, R) = \left(\frac{1}{\lambda} + D\right) e^{\lambda R} \left(e^{\lambda(W+C)} - 1\right). \tag{5.1}$$

Lemma 26 comes from [31, Theorem 1]. It also applies when the segment is not followed by a checkpoint (take $C = 0$). The *slowdown function* is defined as $f(W, C, R) = \frac{\mathbb{E}(W,C,R)}{W}$. We have the following properties:

Lemma 27. *The slowdown function $W \mapsto f(W, C, R)$ has a unique minimum W_{opt} that does not depend on R, is decreasing in the interval $[0, W_{opt}]$ and is increasing in the interval $[W_{opt}, \infty)$.*

Proof. Again, see [31, Theorem 1]. The exact value of W_{opt} is obtained using the Lambert W function, but a first-order approximation is the Young/Daly formula $W_{YD} = \sqrt{\frac{2C}{\lambda}}$. □

Lemma 27 shows that infinite applications should be partitioned into segments of size W_{opt} followed by a checkpoint. What about finite applications? Back to our application $\mathcal{A}$ of duration T_{base}, we partition it into N segments of length W_i, $1 \leq i \leq N$, each followed by a checkpoint C. By linearity of the expectation, the expected time to execute application $\mathcal{A}$ is

$$\mathbb{E}(J) = \sum_{i=1}^{N} \mathbb{E}(W_i, C, R) = \left(\frac{1}{\lambda} + D\right) e^{\lambda R} \sum_{i=1}^{N} \left(e^{\lambda(W_i + C)} - 1\right),$$

where $\sum_{i=1}^{N} W_i = T_{base}$. By convexity of the Exponential function, or by using Lagrange multipliers, we see that $\mathbb{E}(J)$ is minimized when the W_i's take a constant value, i.e., all segments have same length. Thus, we obtain $W_i = \frac{T_{base}}{N}$ for all i, and we aim at finding N that minimizes

$$\mathbb{E}(J) = N\, \mathbb{E}\left(\frac{T_{base}}{N}, C, R\right) = f\left(\frac{T_{base}}{N}, C, R\right) \times T_{base}\ , \tag{5.2}$$

where f is the slowdown function. We let $K_{opt} = \frac{T_{base}}{W_{opt}}$, where W_{opt} achieves the minimum of the slowdown function. K_{opt} would be the optimal number of segments if we could have a non-integer number of segments. Lemma 27 shows that the optimal value N_{ME} of N is either $N_{ME} = \max(1, \lfloor K_{opt} \rfloor)$ or $N_{ME} = \lceil K_{opt} \rceil$, whichever leads to the smallest value of $\mathbb{E}(J)$. In the experiments, we use the simplified Young/Daly expression $N_{ME} = \left\lceil \frac{T_{base}}{W_{YD}} \right\rceil$.

5.3.3 Parallel application and Exponential failure distribution

Because of the memoryless property of the Exponential distribution, the multi-processor case can be reduced to the uni-processor case. Everything holds by replacing the parameter λ by $p\lambda$, where p is the number of processors of application $\mathcal{A}$. To see this formally, say the application $\mathcal{A}$ is executed on p processors $\{P_q\}_{1 \leq q \leq p}$. Let $X_i^q \sim \text{Exp}(\lambda)$, $i \geq 1$, $1 \leq q \leq p$, denote the IID failure IATs on processor P_q. In other words, X_i^q is the random variable for the time between failure number $i-1$ (or the application start if $i = 1$) and failure number i on processor P_q. Let Y_i, $i \geq 1$, denote the failure IATs for (the p processors executing) application $\mathcal{A}$. In other words, Y_i is the random variable for the time between failure number $i-1$ (or the application start if $i = 1$) and failure number i on the whole application.

The assumption $X_i^q \sim \text{Exp}(\lambda)$ formally means that when processor P_q is rejuvenated (or when it is used for the first time), the next failure will strike according to a distribution $\text{Exp}(\lambda)$. If the application starts at time t, and the last failure struck at time $t_1 < t$ on processor P_q, what is the distribution of the probability of the next failure, knowing that P_q has been alive for $t - t_1$ seconds? The memoryless property of Exponential distributions is the key: it is still the same Exponential distribution. To keep notation simple, we let $X_i^q \sim \text{Exp}(\lambda)$, $i \geq 0$, denote the failure IATs on P_q once the application has started (and similarly for Y_i, $i \geq 0$).

First, we have $Y_1 = \min_q(X_1^q)$. Hence $Y_1 \sim \text{Exp}(p\lambda)$ (minimum of p $\text{Exp}(\lambda)$ distributions). Assume that the first failure for application $\mathcal{A}$ stroke at time t_2 (hence $Y_1 = t_2 - t$) on some processor, say processor P_{q_0}, which is then rejuvenated. Because of the memoryless

property, knowing this failure does not change the distribution of the next failure on any other processor, and $Y_2 \sim \text{Exp}(p\lambda)$ for the very same reason that $Y_1 \sim \text{Exp}(p\lambda)$.

5.3.4 Extension without final checkpoint nor initial recovery, and $\mathcal{D}$ Exponential

Consider a parallel application $\mathcal{A}$ with p processors, which is partitioned into segments. This section deals with the case where no checkpoint is enforced at the end of the last segment. By symmetry, we also deal with the case where no recovery is paid when re-executing the first segment after a failure. Let $p\lambda$ denote the failure rate for application $\mathcal{A}$, assuming that application failures obey an Exponential distribution $\mathcal{D} = \text{Exp}(p\lambda)$.

The application is partitioned into N segments of length W_i, with checkpoint cost C_i and recovery cost R_i. Let $C_{tot} = \sum_{i=1}^{N} C_i$ and $R_{tot} = \sum_{i=1}^{N} R_i$. In the model of Sections 5.3.2 and 5.3.3, we had $C_i = C$, $R_i = R$ for $1 \leq i \leq N$, $C_{tot} = NC$, and $R_{tot} = NR$. If no checkpoint is taken after the last segment, $C_N = 0$ and $C_{tot} = (N-1)C$. If no recovery is paid when re-executing the first segment, $R_1 = 0$ and $R_{tot} = (N-1)R$.

What is the optimal strategy (number N of segments and length of each segment) to minimize the expected execution time $\mathbb{E}_N$ of the application? From Lemma 26, we have

$$\mathbb{E}_N = \sum_{i=1}^{N} \mathbb{E}(W_i, C_i, R_i) = \left(\frac{1}{p\lambda} + D\right) \sum_{i=1}^{N} e^{p\lambda R_i} \left(e^{p\lambda(W_i+C_i)} - 1\right),$$

where $\sum_{i=1}^{N} W_i = T_{base}$. Given N, $\mathbb{E}_N$ is minimized when the sum $\sum_{i=1}^{N} e^{p\lambda(W_i+C_i+R_i)}$ is minimized. By convexity of the Exponential function, or by using Lagrange multipliers, we see that $\mathbb{E}_N$ is minimized when the $W_i + C_i + R_i$'s take a constant value W_{seg}. This value is given by

$$NW_{seg} = T_{base} + C_{tot} + R_{tot}, \tag{5.3}$$

and the length W_i of each segment is then computed as $W_i = W_{seg} - C_i - R_i$. If $C_N = 0$, the last segment involves an additional amount C of work duration. Similarly, if $R_1 = 0$, the first segment involves an additional amount R of work duration. Then, we can derive the optimal value of N and W_{seg} as follows: Equation (5.3) gives $N = \frac{T_{base}-R-C+R_1+C_N}{W_{seg}-R-C}$. Plugging this value into

$$\mathbb{E}_N = \left(\frac{1}{p\lambda} + D\right) \left[(N-1)e^{p\lambda R} + e^{p\lambda R_1} + Ne^{p\lambda W_{seg}}\right]$$

enables to solve for W_{seg}, using the Lambert function in a similar way as in [31].

While the derivation is painful, the conclusion is comforting: in the optimal solution, all segments have same length of work, up to an additional recovery for the first segment and an additional checkpoint for the last one. The Young/Daly approximation still holds, as well as all the results of this chapter (whose presentation is much simpler with all segments followed by a checkpoint).

5.3.5 Uni-processor application and arbitrary failure distribution

When failure IATs obey an arbitrary distribution $\mathcal{D}$, they are still IID, because the processor is rejuvenated (replaced by a spare) after each failure. To the best of our knowledge, even the optimal value W_{opt} for the slowdown function is not known analytically. For some distributions, W_{opt} can be computed numerically, using partial moments for the expectation of the time lost due to failures. But note that W_{opt} does not give the optimal checkpointing period for infinite applications, in contrast to the memoryless case. In fact, the optimal checkpointing strategy is not known for infinite applications, let alone for finite applications.

For instance, consider a Weibull distribution $\mathcal{D} \sim \text{WEIBULL}(k, \lambda)$ of shape k and scale λ; its PDF is $\mathbb{P}(X = t) = \frac{k}{\lambda}(\frac{t}{\lambda})^{k-1}e^{-(\frac{t}{\lambda})^k}$ for $t > 0$. If $k < 1$, the instantaneous failure rate of $\mathcal{D}$ is decreasing with time (infant mortality), checkpoints should be spaced more and more as time progresses since the last failure. On the contrary, if $k > 1$, the instantaneous failure rate of $\mathcal{D}$ is increasing with time (ageing) and, hence, checkpoints should be spaced less and less. This explains that the optimal checkpointing strategy is aperiodic. See [82, 113, 129] for more details.

5.3.6 Parallel application and arbitrary failure distribution

When $\mathcal{D}$ is arbitrary, even though the failure IATs X_i^q are IID on each processor, they are not for (the p processors executing) application $\mathcal{A}$. In other words, the Y_i are not IID, unless $\mathcal{D}$ is Exponential. However, owing to the theory on the superposition of renewal processes, whenever $\mathcal{D}$ is continuous and of finite expectation μ_{ind}, we know that the following limit exists, where n is the number of failures:

$$\lim_{n\to\infty} \mathbb{E}\left(\frac{\sum_{i=1}^{n} Y_i}{n}\right) = \frac{\mu_{ind}}{p}. \tag{5.4}$$

This result is given as Formula 1.4 in [103]. See also [80] for an elementary proof using Wald's equation. Equation (5.4) is good news because we can define the application MTBF as $\frac{\mu_{ind}}{p}$: in average, a failure will strike the application every $\frac{\mu_{ind}}{p}$ seconds. Note that the MTBF is given a new definition here: the failures striking the parallel application $\mathcal{A}$ are not IID, so there is no longer a mean time before the next failure of the application. Instead, there is a limit on the average time between failures.

At any time, the distribution of the next failure is complicated because it must account for the history of the $p-1$ processors that have not been rejuvenated when the last failure stroke. Indeed, assume that the execution of application $\mathcal{A}$ was started on p fresh processors $\{P_q\}_{1\leq q\leq p}$, and that the last failure stroke on processor P_q at time t_q (where $t_q = 0$ if P_q has never been struck). Let $i(q)$ be the index of the last failure on P_q (where $i(q) = 0$ if P_q has never been struck). To simplify notation, say that the last failure stroke processor P_1, meaning that $t_q < t_1$ for $q \geq 2$. Now for $q \geq 2$, the probability that the next failure on P_q strikes at time t (it will be failure number $i(q) + 1$ for P_q) is

$$\mathbb{P}(X_{i(q)+1}^q = t | X_{i(q)+1}^q \geq t - t_q).$$

In other words, only $X_{i(1)+1}^1$ follows the distribution $\mathcal{D}$, while each $X_{i(q)+1}^q$, $q \geq 2$, is shifted. To compute the distribution of the next failure of application $\mathcal{A}$, we need to compute the distribution of the minimum of all the $X_{i(q)+1}^q$'s, which are not identical because of their history.

There is a theoretical approach that simplifies the problem, namely rejuvenating all the p processors of the application after each failure (and before starting the execution of the application). Of course, this is impossible in practice when p exceeds a small number, but it is nice from a theory perspective: with total rejuvenation, each failure becomes a renewal point for the whole application, and the failure IATs that strike the application become IID: their distribution is the minimum of p IID distributions $\mathcal{D}$. Even better, there are a few failure distributions $\mathcal{D}$ such that the minimum of p IID instances also obey the same distribution $\mathcal{D}$ (with scaled parameters). For instance, consider a Weibull distribution $\mathcal{D} \sim \text{WEIBULL}(k, \lambda)$ of shape k and scale λ, whose expectation is $\mu_{ind} = \lambda\Gamma(1 + \frac{1}{k})$, where Γ denotes the Gamma function $\Gamma(t) = \int_0^\infty x^{t-1}e^{-x}dx$ for $t > 0$. Then the minimum Y of p IID $\text{WEIBULL}(k, \lambda)$ is also a Weibull distribution $Y \sim \text{WEIBULL}(k, \frac{\lambda}{p^{1/k}})$. We observe

that the MTBF does not scale linearly with p, unless $k = 1$: the expectation of Y is $\mu = \frac{\mu_{ind}}{p^{1/k}}$. This discussion explains the errors in [64]: (i) the platform cannot obey a Weibull distribution, unless total rejuvenation is used; and (ii) assuming total rejuvenation, the MTBF of an application is not inversely proportional to its number of processors.

A realistic approach to cope with the not-IID problem is to discretize time into small quanta, and to use dynamic programming to compute the best checkpoint strategy for application $\mathcal{A}$ up to the next failure [31]. Obviously, the smaller the quanta, the more accurate the results, but the more costly the dynamic programming algorithm. The approach in [31] greedily uses this strategy from one failure to another, up to the completion of the application. However, optimizing checkpoints up to the next failure, while optimal from one failure to the next (up to the precision of the quanta), may well be sub-optimal for the whole application. A main contribution of this chapter is to introduce a new greedy strategy and to prove an approximation bound for its performance. To the best of our knowledge, this is the first theoretical result for parallel applications with non-memoryless failures.

5.4 The NextStep heuristic

In this section, we present the NextStep heuristic to checkpoint parallel applications under any failure probability distribution. The main idea of NextStep is the following: after each failure, NextStep is able to find the checkpointing strategy that maximizes the expected efficiency (see below) before the next failure or the end of the application. Intuitively, optimizing the expected efficiency on a *failure-by-failure* basis should lead to a good approximation on the optimal solution, at least for large application sizes. One major contribution of this work is to show that NextStep is asymptotically optimal for arbitrary failure distributions.

We first introduce notation in Section 5.4.1, before formally describing NextStep in Section 5.4.2. Finally, we prove the asymptotic optimality in Section 5.4.3.

5.4.1 Preliminaries

Consider a parallel application $\mathcal{A}$ of length T_{base} executing on p processors, with checkpoint time C. We define one unit of work of the application$\mathcal{A}$ as the parallel work executed within one second, so that we can express application progress either in seconds or in work units, whichever is more natural for the reader. Assume that the application just experienced a failure, and it is ready to resume execution of the remaining W seconds of work (or the application is just starting, and then $W = T_{base}$). For any processor P_j, $1 \leq j \leq p$, let τ_j be the time elapsed since its last failure. In particular, if P_j has been hit by the last failure, then $\tau_j = 0$; note also that we do not assume fresh processors when starting the application. We call $\vec{\tau} = (\tau_1, \tau_2, \ldots, \tau_p)$ the *history* vector.

Given a checkpointing strategy, an application with W remaining work units and a history vector $\vec{\tau}$, the function $first(W|\vec{\tau})$ returns the size W_{first} of the segment preceding the first checkpoint.

Work. Let $\mathcal{W}(W|\vec{\tau})$ be the random variable that quantifies the number of work units

successfully executed before the next failure. We have the following recursion:

$$\mathcal{W}(0|\vec{\tau}) = 0$$
$$\mathcal{W}(W|\vec{\tau}) = \begin{cases} W_{first} + \mathcal{W}(W - W_{first}|\vec{\tau} + \overrightarrow{W_{first} + C}) \\ \quad \text{if the } p \text{ processors do not fail during} \\ \quad \text{the next } W_1 + C \text{ units of time,} \\ 0 \quad \text{otherwise.} \end{cases} \tag{5.5}$$

In Equation (5.5), given a scalar quantity x, $\overrightarrow{x} = (x, x, \ldots, x)$ denotes the vector with p identical components equal to x. Weighting the two cases in Equation (5.5) by their probabilities of occurrence, we obtain the expected number of work units successfully computed before the next failure:

$$\mathbb{E}(\mathcal{W}(W|\vec{\tau})) = \mathbb{P}_{suc}(W_{first} + C|\vec{\tau})(W_1 + \mathbb{E}(\mathcal{W}(W - W_{first}|\vec{\tau} + \overrightarrow{W_{first} + C}))), \tag{5.6}$$

where the probability of success $\mathbb{P}_{suc}$ is computed as

$$\mathbb{P}_{suc}(x|\vec{\tau}) = \prod_{i=1}^{p} \mathbb{P}(X \geq x + \tau_i | X \geq \tau_i). \tag{5.7}$$

X is a generic random variable following the probability distribution $\mathcal{D}$, the failure inter-arrival time on each processor. Given any such distribution $\mathcal{D}$, $\mathbb{P}_{suc}(x|\vec{\tau})$ can be computed in time $O(p)$.

Efficiency. Rather than focusing solely on the work done, we aim at maximizing the expected efficiency, which also depends on the number of checkpoints that have been taken. This is particularly crucial at the end of the application, where maximizing the number of work units until the next failure may not be the best strategy if the application is about to complete. Indeed, the efficiency also depends on the time spent computing; if no failures occur, it depends on the number of checkpoints that are taken. Hence, we define $\mathbb{E}_W(W, \vec{\tau}, N)$ as the maximum expected number of work units until the next failure (or the completion of the application if no failure occurs) using N checkpoints; similarly, $\mathbb{E}_{T_{next}}(W, \vec{\tau}, N)$ is the expected time until the next failure, or before the completion of the application if no failure occurs. Note that the number N of checkpoints only matters in the latter case where the application has completed.

Finally, if an application still needs to be processed for W units of time, with a history $\vec{\tau}$, we define the maximum possible efficiency among all possible numbers of checkpoints N:

$$\mathbb{E}_e(W, \vec{\tau}) = \max_N \frac{\mathbb{E}_W(W, \vec{\tau}, N)}{\mathbb{E}_{T_{next}}(W, \vec{\tau}, N)}. \tag{5.8}$$

Time discretization. We introduce a time quantum u, and discretize time into quanta. This means that all quantities (segment sizes, checkpoint and recovery times) are integer multiples of u, and that failures strike at the end of a quantum. More precisely, the probability that a failure happens at the end of quantum i is $\int_{(i-1)u}^{iu} f(x|\vec{\tau})dx$, where $f(x|\vec{\tau})$ is the probability density function of the platform failure distribution $\mathcal{D}$ in the continuous case conditioned by the history. This discretization restricts the search for an optimal execution to a finite set of possible executions. The trade-off is that a smaller value of u leads to a more accurate solution, but also to a higher number of states in the algorithm, hence, to a higher computing time.

In what follows, if a variable y is defined in work units, $y^* = y/u$ is the corresponding number of quanta, which we always suppose integer. For instance, the application size becomes $W^* = W/u$ and the checkpoint size $C^* = C/u$. Similarly, we let $\mathbb{P}^*_{suc}(x^*|\vec{\tau^*}) \triangleq \mathbb{P}_{suc}(ux|u\vec{\tau})$ be the probability that the next x^* quanta succeed given the history $\vec{\tau^*}$, expressed in number of quanta.

Algorithm 5: $compE(x^*, N_p, N_f)$

```
1  if x* = 0 then return 0;
2  if N_f = 1 then
3      τ⃗* ← τ⃗*_0 + overrightarrow(W* − x* + N_p C*);
4      best ← x* P*_suc(x* + C* | τ⃗*) ;
5      solve[x*][N_p][N_f] ← (best, x*);
6      return best;
7  if solve[x*][N_p][N_f] = (best, W*_first) then return best;
8  else
9      best ← −∞;
10     τ⃗* ← τ⃗*_0 + overrightarrow(W* − x* + N_p C*);
11     for i = 1 to x* do
12         work ← compE(x* − i, N_p + 1, N_f − 1);
13         cur ← P*_suc(i + C* | τ⃗*) × (i + work);
14         if cur > best then
15             best ← cur;
16             W*_first ← i;
17     solve[x*][N_p][N_f] ← (best, W*_first);
18     return best;
```

5.4.2 NextStep

We define NextStep formally as: find a function returning the size of the first segment to be checkpointed, such that $\mathbb{E}_e(W, \vec{\tau_0})$ is maximized. Here, $\vec{\tau_0}$ corresponds to the initial history of the platform when the execution starts. Solving NextStep analytically seems out of reach, but the recursive definition of $\mathbb{E}(\mathcal{W}(W|\vec{\tau}))$ (see Equation (5.6)), together with time discretization, allows us to compute the maximum efficiency. Indeed, there is no need to keep the time elapsed since the last failure of each processor as a parameter of the recursive calls. This is because the τ variables of all processors evolve identically: recursive calls only correspond to cases where no failure has occurred, hence the same quantity is added to the history of each processor. The algorithm is called again each time a failure occurs, to decide where checkpoints should be taken.

Thanks to the discretization, all the $\mathbb{E}_W(W, \vec{\tau_0}, N)$ values can be computed with a time quantum u. We let x^* be the number of quanta that remain to proceed (where initially, $x^* = W^*$). We need to find and store the best solutions for any possible values of x^* and N in the recursive call. Hence, we further consider N_p, the number of checkpoints already taken, and N_f, the number of checkpoints that can still be taken (where $N_p + N_f = N$). This corresponds to Algorithm 5: the *compE* procedure fills a table *solve* that contains, for any triple (x^*, N_p, N_f), the maximum expected work duration until the next failure for these parameters, and the best segment size W^*_{first} that achieves this. For $N_f = 1$, the only possibility is to compute x^* in its entirety and then checkpoint. Otherwise, we try all possible places for the first checkpoint, and recursively call *compE*. If a value with a given (x^*, N_p, N_f) had been computed before, we retrieve the corresponding result line 7.

There remains to compute $\mathbb{E}_{T_{next}}(W^*, \vec{\tau^*}, N)$, i.e., the expected time until next failure or application completion. The following lemma helps us compute these values efficiently with discrete segments:

Lemma 28. *Using discrete quanta of size u, the expectation of the time before the next*

Algorithm 6: NEXTSTEP (W^*)

```
   /* Compute E_{T_next}(W*, τ⃗*_0, n_c) for n_c ∈ [1, W*]                    */
1  S ← 0;
2  for i = 0 to W* − 1 do S ← S + P*_suc(i|τ⃗*_0);
3  for n_c = 1 to W* do
4  |  for i = 1 to C* do
5  |  |  S ← S + P*_suc(W* + (n_c − 1)C* + i|τ⃗*_0);
6  |  E_{T_next}(W*, τ⃗*_0, n_c) ← S;

   /* Compute E_W(W*, τ⃗*_0, n_c) (array solve)                              */
7  for n_c = 1 to W* do compE(W*, 0, n_c);

   /* Solution of NEXTSTEP                                                   */
8  best ← −∞; N_c ← 0; W*_first ← 0;
9  for n_c = 1 to W* do
10 |  cur ← arg_first(solve[W*][0][n_c])/E_{T_next}(W*, τ⃗*_0, n_c);
11 |  cursegment ← arg_second(solve[W*][0][n_c]);
12 |  if cur > best then
13 |  |  best ← cur; N_c ← n_c; W*_first ← cursegment;
14 return (N_c, W*_first);
```

failure or the completion of the application, expressed in number of quanta, is the following:

$$\mathbb{E}_{T_{next}}(W^*, \vec{\tau_0^*}, N) = \sum_{i=0}^{W^*+NC^*-1} \mathbb{P}^*_{suc}(i|\vec{\tau_0^*}).$$

Proof. Let X denote the random variable of the number of quanta executed before the next failure (or the completion of the application) given the history $\vec{\tau_0^*}$, the total number of quanta of the application W^* and the number of checkpoints N. Clearly, X is taking integer values in $[1, W^* + NC^*]$, thus

$$\begin{aligned}\mathbb{E}(X) &= \sum_{i=1}^{W^*+NC^*} i\mathbb{P}\{X = i\} = \sum_{i=1}^{W^*+NC^*} \mathbb{P}\{X \geq i\} \\ &= \sum_{i=1}^{W^*+NC^*} \mathbb{P}^*_{suc}(i-1|\vec{\tau_0^*}) = \sum_{i=0}^{W^*+NC^*-1} \mathbb{P}^*_{suc}(i|\vec{\tau_0^*}).\end{aligned}$$

□

From Lemma 28, we derive that:
$\mathbb{E}_{T_{next}}(W^*, \vec{\tau_0^*}, n_c + 1) = \mathbb{E}_{T_{next}}(W^*, \vec{\tau_0^*}, n_c) + \sum_{i=W^*+n_cC^*}^{W^*+(n_c+1)C^*-1} \mathbb{P}^*_{suc}(i|\vec{\tau_0^*})$.
This is used in Algorithm 6 to compute all the $\mathbb{E}_{T_{next}}$ values more efficiently on lines 1–6. Algorithm 5 is called to fill the *solve* table with all values of $\mathbb{E}_W$ in line 7. We obtain the efficiency $\mathbb{E}_e(W^*, \vec{\tau_0^*})$ for all possible number of checkpoints and keep the maximum, see lines 8–13. Finally, the algorithm returns the values for N and W^*_{first} corresponding to the maximum efficiency, which allows us to rebuild completely the corresponding solution using the table *solve*.

Proposition 1. *Using a time quantum u, and for any failure inter-arrival time distribution, Algorithm 6 computes the solution to* NEXTSTEP *(maximizing efficiency) in time $O(p(W^*)^4)$.*

Proof. The NextStep algorithm consists of three steps. In the first step, it computes all values of $\mathbb{E}_{T_{next}}(W^*, \vec{\tau_0^*}, n_c)$ for $n_c \in [1, W^*]$. To do so, two loops are executed. The first one has W^* steps, where each step computes a single addition. The value of $\mathbb{P}^*_{suc}(W^* + (n_c - 1)C^* + i|\vec{\tau_0^*})$ is the product of the individual probability of failure of the p processors (as in Equation (5.7)). We assume that the individual probabilities of failure can be computed in $O(1)$, thus the loop takes a time $O(pW^*)$. The second loop is similar, with two nested loops, and its complexity is $O(pW^*C^*)$. We can safely assume that $C^* \leq W^*$, otherwise doing any checkpoint is straightforwardly bad (if we succeed the checkpoint, we would have succeeded the entire application), and the complexity is at most $O(p(W^*)^2)$.

In the second step, the algorithm fills the table *solve* and calls $compE(W^*, 0, n_c)$ for $n_c \in [1, W^*]$. The function $compE(x, y, z)$ fills the table entry corresponding to its parameters if necessary, with eventual recursive calls to *compE* where $y + z$ is constant and x decreases. Given the initial calls with $x = W^*$ and $y + z \in [1, W^*]$ the number of entries written in the table is at most $\frac{W^{*3}}{2}$. To upper bound the overall complexity of this step, we first note that an entry may only be written in the table if the *compE* function is called with the same parameters. In the sub-case $N_f = 1$, this takes few operations and involves a call to $\mathbb{P}^*_{suc}$, thus a time $O(p)$. Otherwise, this means *compE* has been called with parameters corresponding to the last sub-case. If so, a loop is executed $x^* \leq W^*$ times, and each iteration requires a call to *compE*, which either takes time $O(1)$ or fills another entry of the table (therefore the complexity is taken into account for this other entry), and the other operations are in $O(1)$, except the computation of $\mathbb{P}^*_{suc}$ in $O(p)$. Overall, the individual cost of each entry of the table is at most in $O(pW^*)$. Finally the calls to *compE* that do not fill the table may only be made recursively and were taken into account in the analysis. Given the size of the table in $O((W^*)^3)$, this second step is in $O(p(W^*)^4)$.

Finally, the last step returning the solution consists of a loop with W^* iterations, where each iteration is done in $O(1)$, which gives a complexity in $O(W^*)$. The complexity is dominated by the second step; hence, the result. □

5.4.3 Asymptotic analysis

We proceed in two steps. First, we show that for an infinite application and for any integer n, the expected work completed by NextStep before the n-th failure (which happens at a random time) is larger than or equal to the one of any other strategy. Then, we show that the expected work processed within T units of time is asymptotically optimal with T (Theorem 20).

Expected work completed before the n-th failure

We compare NextStep to any other strategy for an infinite application that has an infinite number of failures. We assume that the application starts at time $t_0 = 0$. For $k \geq 1$, let t_k be the random variable representing the date of the k-th failure, and for $k \geq 0$, let $\vec{\tau}_k$ be the random variable representing the history of the application at time t_k. Note that neither t_k nor $\vec{\tau}_k$ depend on the checkpointing strategy. For any $n \geq 1$, let WF_n be the random variable corresponding to the expected work executed from the start up to failure n by NextStep, and let OPT_n denote the same variable for an optimal strategy, both depending on the initial history $\vec{\tau}_0$. We show that $\mathbb{E}(WF_n) \geq \mathbb{E}(OPT_n)$.

We define wf_k (resp. opt_k) the random variable representing the work executed by NextStep (resp. an optimal strategy) between times t_{k-1} and t_k. At any decision point, i.e., after each failure, the expected time before the next failure or the completion is in fact

the expected time before the next failure, because the application is infinite; hence, this time does not depend upon the number of checkpoints. From Equation (5.8), we deduce that, for any history, NextStep maximizes the expected work accomplished before the next failure. Hence, we have for any $k \geq 1$ and any possible history $\vec{\tau}$ at the $(k-1)$-th failure (or $\vec{\tau} = \vec{\tau_0}$ if $k = 1$):

$$\mathbb{E}(wf_k | \vec{\tau}_{k-1} = \vec{\tau}) \geq \mathbb{E}(opt_k | \vec{\tau}_{k-1} = \vec{\tau}).$$

This inequality holds for any possible history $\vec{\tau}$, i.e., for any possible event $\{\vec{\tau}_{k-1} = \vec{\tau}\}$, and $\vec{\tau}_{k-1}$ does not depend on the strategy. Therefore, this inequality can be directly extended to $\mathbb{E}(wf_k | \vec{\tau}_{k-1})$ and $\mathbb{E}(opt_k | \vec{\tau}_{k-1})$. Note that these expectations are conditioned by a random variable instead of an event, thus are random variables themselves (constant for $k = 1$ if we consider $\vec{\tau_0}$ as a constant random variable) which always verify $\mathbb{E}(wf_k | \vec{\tau}_{k-1}) \geq \mathbb{E}(opt_k | \vec{\tau}_{k-1})$. In particular:

$$\mathbb{E}(\mathbb{E}(wf_k | \vec{\tau}_{k-1})) \geq \mathbb{E}(\mathbb{E}(opt_k | \vec{\tau}_{k-1})).$$

This result can be combined with the property $\mathbb{E}(X) = \mathbb{E}(\mathbb{E}(X|Y))$ (Law of Total Expectation) whenever both sides exist [149, p. 179] to obtain, for all $k \geq 1$,

$$\mathbb{E}(wf_k) = \mathbb{E}(\mathbb{E}(wf_k | \vec{\tau}_{k-1})) \geq \mathbb{E}(\mathbb{E}(opt_k | \vec{\tau}_{k-1})) = \mathbb{E}(opt_k).$$

Finally, we obtain:

$$\begin{aligned} \mathbb{E}(WF_n) &= \mathbb{E}\left(\sum_{k=1}^{n} wf_k\right) = \sum_{k=1}^{n} \mathbb{E}(wf_k) \\ &\geq \sum_{k=1}^{n} \mathbb{E}(opt_k) = \mathbb{E}\left(\sum_{k=1}^{n} opt_k\right) = \mathbb{E}(OPT_n). \end{aligned}$$

This shows that the expected work completed by NextStep before the n-th failure is larger than or equal to the one of any other strategy.

NextStep is asymptotically optimal

For any T, we show how to define $n(T)$, the index of a failure striking at a time close enough to T, so that the relative work difference performed between T and $t_{n(T)}$ (whichever comes first) is negligible:

Lemma 29. *Let* $n(T) = p\lfloor \frac{T}{\mathbb{E}(X)} \rfloor$, *then* $\lim_{T \to \infty} \frac{\mathbb{E}(|T - t_{n(T)}|)}{T} = 0$.

Proof. Consider an infinitely long application executing on p processors P_i, $1 \leq i \leq p$. Let X denote the random variable for failure IATs on each processor if there is no history. For $T > 0$, we fix $K(T) = \left\lfloor \frac{T}{\mathbb{E}(X)} \right\rfloor$, thus $n(T) = pK(T)$.

Let $t_{i,j}$ be the random variable representing the time when processor P_i fails for the j-th time. Clearly, for all i and $k > 0$, $t_{i,k+1} - t_{i,k}$ follows the distribution X, because P_i is rejuvenated after each failure. Therefore, $\mathbb{E}(t_{i,j}) = \mathbb{E}(X_{i,0}) + (j-1)\mathbb{E}(X)$, where $\mathbb{E}(X_{i,0})$ depends on the initial state of processor P_i. We then use a variant of the strong law of large numbers [45, Ex. 8 p. 137]: If $(X_1, X_2, \ldots, X_j)$ are identically distributed with finite expectations and $S_j = \sum_{k=1}^{j} X_k$, then $\frac{S_j}{j} \to \mathbb{E}(X_1)$ in L^1, i.e., $\lim_{j \to \infty} \mathbb{E}\left(\left|\frac{S_j}{j} - \mathbb{E}(X_1)\right|\right) = 0$. Applying this result with $X_k = t_{i,k+1} - t_{i,k}$, we obtain $S_{j-1} = t_{i,j} - t_{i,1}$ and

$$\forall i, \lim_{j \to \infty} \mathbb{E}\left(\left|\frac{t_{i,j} - t_{i,1}}{j-1} - \mathbb{E}(X)\right|\right) = 0.$$

For any given j, since the $t_{i,j} - t_{i,1}$'s are identically distributed for all i, we can define a function $\epsilon(j)$ verifying $\lim_{j\to\infty} \epsilon(j) = 0$ and such that, using triangular inequalities:

$$\mathbb{E}(|t_{i,j} - j\mathbb{E}(X)|) \leq j\epsilon(j) + \mathbb{E}(X) + t_{i,1}$$

for all i and j. Finally, $\min_{1\leq i\leq p} t_{i,K(T)} \leq t_{n(T)} \leq \max_{1\leq i\leq p} t_{i,K(T)}$, because the total number of failures is the sum of the number of failures of each processor and $n(T) = pK(T)$. Hence,

$$\begin{aligned}\mathbb{E}(|t_{n(T)} - K(T)\mathbb{E}(X)|) &\leq \mathbb{E}(\max_{1\leq i\leq p}(|t_{i,K(T)} - K(T)\mathbb{E}(X)|))\\ &\leq \mathbb{E}(\sum_{i=1}^{p}(|t_{i,K(T)} - K(T)\mathbb{E}(X)|))\\ &\leq pK(T)\epsilon(K(T)) + p\mathbb{E}(X) + \sum_{i=1}^{p} t_{i,1}.\end{aligned}$$

By definition, $K(T)\mathbb{E}(X) \leq T \leq (K(T)+1)\mathbb{E}(X)$, and, because of the triangular inequality, we have:

$$\begin{aligned}\frac{\mathbb{E}(|T - t_{n(T)}|)}{T} &= \frac{\mathbb{E}(|(T - K(T)\mathbb{E}(X)) + (K(T)\mathbb{E}(X) - t_{n(T)})|)}{T}\\ &\leq \frac{\mathbb{E}(X)}{K(T)\mathbb{E}(X)} + \frac{K(T)p\epsilon K(T)}{K(T)\mathbb{E}(X)} + \frac{p\mathbb{E}(X)}{K(T)\mathbb{E}(X)} + \frac{\sum_{i=1}^{p} t_{i,1}}{K(T)\mathbb{E}(X)}\\ &= \frac{1}{K(T)} + \frac{p}{\mathbb{E}(X)}\epsilon(K(T)) + \frac{p}{K(T)} + \frac{\sum_{i=1}^{p} t_{i,1}}{K(T)\mathbb{E}(X)}.\end{aligned}$$

Here, p, $\mathbb{E}(X)$, and $\sum_{i=1}^{p} t_{i,1}$ are fixed, while $\lim_{T\to\infty} K(T) = \infty$. Hence the result, using $\lim_{K(T)\to\infty} \epsilon(K(T)) = 0$. □

Theorem 20. *For any T, with $n(T) = p\lfloor\frac{T}{\mathbb{E}(X)}\rfloor$, let $w^*_{n(T)}$ be the optimal expected work done up to time $t_{n(T)}$ (from the start to the $n(T)$-th failure). Then, for any checkpointing strategy, we have $\frac{\mathbb{E}_W([0,T])}{T} \leq \frac{w^*_{n(T)}}{T} + o(1)$. Furthermore, with NextStep, we have $\frac{\mathbb{E}_W([0,T])}{T} \geq \frac{w^*_{n(T)}}{T} - o(1)$. Hence, NextStep is asymptotically optimal.*

Proof. Assuming $\mathbb{E}_W([a,b]) = 0$ if $a > b$, thanks to Lemma 29, we obtain that, for any strategy,

$$\begin{aligned}\frac{\mathbb{E}_W(T)}{T} &\leq \frac{\mathbb{E}_W([0,t_{n(T)}])}{T} + \frac{\mathbb{E}_W([t_{n(T)},T])}{T}\\ &\leq \frac{w^*_{n(T)}}{T} + \frac{\mathbb{E}(|T - t_{n(T)}|)}{T} = \frac{w^*_{n(T)}}{T} + o(1).\end{aligned}$$

Furthermore, for NextStep, we have:

$$\begin{aligned}\frac{\mathbb{E}_W(T)}{T} &\geq \frac{\mathbb{E}_W([0,t_{n(T)}])}{T} - \frac{\mathbb{E}_W([T,t_{n(T)}])}{T}\\ &\geq \frac{w^*_{n(T)}}{T} - \frac{\mathbb{E}(|T - t_{n(T)}|)}{T} = \frac{w^*_{n(T)}}{T} - o(1),\end{aligned}$$

which concludes the proof. □

Counter-example to optimality

The NextStep heuristic is asymptotically optimal but not always optimal. This is because, for short applications, maximizing the efficiency until the next failure is not exactly equivalent to minimizing the makespan. An example where NextStep is not optimal, for an Exponential distribution is omitted here for this thesis conciseness but can be found in [R3, Appendix A]. The example is designed as a worst-case scenario and shows that the number of checkpoints may differ between NextStep and the optimal.

5.5 On the dynamic version of the optimal static strategy for an Exponential distribution

Sections 5.3.2 and 5.3.3 have shown how to statically compute the optimal strategy to minimize the expected makespan of an application when the failures obey an Exponential distribution. This optimal strategy is *static*, meaning that we compute the number and length of the application segments once and for all, before starting the execution. On the contrary, the NextStep strategy is dynamic, since it is called after each failure. One may envision a dynamic version of the optimal static strategy, where one would recompute the number and length of the application segments after each failure (and maybe after each checkpoint too), as a function of the remaining size of the application. Here we show that this dynamic approach is identical to the static one. This new result demonstrates the fairness of comparing NextStep with a static approach.. We start with a few notations before formally stating this result.

5.5.1 Notations

In the following, we consider a sequential or parallel application of length T_{base}. A checkpointing strategy $\mathcal{S}$ is defined as $\mathcal{S} = \{c_1, c_2, \ldots, c_m\}$, where each $c_k \in (0, T_{base})$ denotes the amount of the work executed until checkpoint number k. Note that we assume that there is a checkpoint at the end, i.e., $c_m = T_{base}$.

When a failure occurs, let $\mathbb{E}(R)$ be the expected time before the processors are ready to work again. This includes the downtime and a recovery time, but may be longer if we encounter another failure during the recovery time[3]. For a given checkpointing strategy $\mathcal{S}$ and a work $w \in \mathcal{S} \cup \{0\}$, we denote by $\mathbb{E}([0, w], \mathcal{S})$ the expected time between the start of the application and the completion of the checkpoint corresponding to w units of work. Similarly, we denote by $\mathbb{E}([w, T_{base}], \mathcal{S})$ the expected time between the moment the checkpoint corresponding to w units of work is completed (or the start of the application if $w = 0$) and the moment the application finishes the completion, including the last checkpoint. If we do not have $w \in \mathcal{S} \cup \{0\}$, both expectations are considered infinite. With these definitions, we clearly have:

$$\forall w \in \mathcal{S}, \mathbb{E}([0, T_{base}], \mathcal{S}) = \mathbb{E}([0, w], \mathcal{S}) + \mathbb{E}([w, T_{base}], \mathcal{S}).$$

Finally, given a work $w \in [0, T_{base}]$, we let $\mathcal{S}^*_w$ be a checkpointing strategy such that for all $\mathcal{S}$, we have $\mathbb{E}([w, T_{base}], \mathcal{S}) \geq \mathbb{E}([w, T_{base}], \mathcal{S}^*_w)$. Although multiple checkpointing strategies may minimize this expectation, the value of this expectation $\mathbb{E}^*_w \triangleq \mathbb{E}([w, T_{base}], \mathcal{S}^*_w)$ is unique and well defined. Intuitively, $\mathcal{S}^*_w$ is an optimal checkpointing strategy for the end of the application after w units have been processed and checkpointed.

[3]Similarly to Equation (5.1), we have $\mathbb{E}(R) = De^{\lambda R} + \frac{1}{\lambda}\left(e^{\lambda R} - 1\right)$. But we do not need to know the value of $\mathbb{E}(R)$ in this derivation.

5.5.2 Main result

Theorem 21. *For an application of length T_{base}, consider the following two approaches:*

- *(A) Static Strategy: Find an optimal checkpointing heuristic $\mathcal{S}_0^*$ that minimizes the total expected makespan $\mathbb{E}_0^*$ and does not update the strategy until the application is completed.*
- *(B) Dynamic Strategy: Start with the best static strategy $\mathcal{S}_0^*$, then whenever an **event** occurs, i.e., a segment is completed or a failure happens, find an optimal static checkpointing strategy minimizing the remaining expected makespan. If the remaining expected makespan is strictly smaller with the new strategy, update the checkpointing strategy accordingly.*

The static strategy (A) and the dynamic strategy (B) are identical.

The optimal static strategy (A) is well-known and uses N_{ME} segments, where N_{ME} is given in Section 5.3.2. The value of N_{ME} depends upon the length of the application that remains to be processed, so strategy (B) could compute a different value when called after the first checkpoint or the first failure. The proof shows that this is never the case.

Proof. Initially, both strategies are identical by definition. We assume that the strategy (B) can diverge from the strategy (A) and obtain a contradiction. Suppose that both strategies are different. Then, there exists a failure scenario in which both strategies diverge. Consider such scenario and let W be the total work executed and backed-up when the first *event* e occurs, after which strategy (B) becomes different from strategy (A).

After this event e, the expected resulting makespan of strategy(A) is $\mathbb{E}_{remain}^A = t(e) + \mathbb{E}([W, T_{base}], \mathcal{S}_0^*)$, where $t(e) = 0$ if the event is the end of a segment, $t(e) = D$ if the event is a failure in the first segment for the model in which a recovery is not necessary for the first segment, and $t(e) = \mathbb{E}(R)$ otherwise. In any case, $t(e)$ is a cost in execution time independent to the checkpointing heuristic. We must finish the processing of the as planned with strategy $\mathcal{S}_0^*$. The latter is identical to $\mathbb{E}([W, T_{base}], \mathcal{S}_0^*)$, because the distribution is memoryless, therefore we are exactly at the same point after the recovery as we were when we first succeeded the checkpoint corresponding to W units of work.

Strategy (B) also needs to spend $t(e)$ units of time to deal with the event and is able to find a new checkpointing strategy such that the overall expected remaining makespan is reduced. As before, because the distribution is memoryless, an optimal strategy is $\mathcal{S}_W^*$. By assumption, this new strategy reduces the total expected makespan. Thus,

$$\begin{aligned}
&\mathbb{E}(R) + \mathbb{E}([W, T_{base}], \mathcal{S}_W^*) = \mathbb{E}_{remain}^B < \mathbb{E}_{remain}^A \\
&\mathbb{E}(R) + \mathbb{E}([W, T_{base}], \mathcal{S}_W^*) < \mathbb{E}(R) + \mathbb{E}([W, T_{base}], \mathcal{S}_0^*)
\end{aligned}$$

Finally,

$$\mathbb{E}([W, T_{base}], \mathcal{S}_W^*) < \mathbb{E}([W, T_{base}], \mathcal{S}_0^*).$$

Now, suppose that we had applied the strategy $S_2 = (\mathcal{S}_0^* \setminus [W, T_{base}]) \cup (\mathcal{S}_W^* \setminus (0, W))$. The total expectation would have been:

$$\begin{aligned}
\mathbb{E}([0, T_{base}], \mathcal{S}_2) &= \mathbb{E}([0, W], \mathcal{S}_2) + \mathbb{E}([W, T_{base}], \mathcal{S}_2) \\
&= \mathbb{E}([0, W], \mathcal{S}_0^*) + \mathbb{E}([W, T_{base}], \mathcal{S}_W^*) \\
\mathbb{E}([0, T_{base}], \mathcal{S}_2) &< \mathbb{E}([0, W], \mathcal{S}_0^*) + \mathbb{E}([W, T_{base}], \mathcal{S}_0^*) \\
\mathbb{E}([0, T_{base}], \mathcal{S}_2) &< \mathbb{E}([0, T_{base}], \mathcal{S}_0^*) \\
\mathbb{E}([0, T_{base}], \mathcal{S}_2) &< \mathbb{E}_0^*
\end{aligned}$$

This contradicts the definition of $\mathcal{S}_0^*$ and $\mathbb{E}_0^*$. □

5.6 Performance Evaluation

In this section, we evaluate and compare the performance of NextStep with the Young/-Daly periodic checkpointing heuristic, using simulations on synthetic applications with various parameters, and subject to failures that are sampled from a wide range of probability distributions. Section 5.6.1 details application parameters and failure distributions. Section 5.6.2 presents all simulation results.

To the best of our knowledge, the Young/Daly periodic checkpointing heuristic is the only competitor to our new approach. Indeed, we deal with parallel applications, arbitrary failure probability distributions, and after a failure we only rejuvenate the faulty processor. While some strategies have been derived for Weibull probability distributions [82, 113, 129], they either assume a single processor or total rejuvenation, so that application IATs always obey a scaled Weibull distribution. As already mentioned, total rejuvenation is not an option for HPC platforms.

5.6.1 Simulation Setup

Algorithms

We compare the performance of NextStep with YoungDaly, the Young/Daly periodic checkpointing strategy (see Sections 5.3.2 and 5.3.3). For YoungDaly, an application $\mathcal{A}$ of length T_{base} and using p processors is divided into $N_{ME} = \left\lceil \frac{T_{base}}{W_{YD}} \right\rceil$ equal-size segments, each followed by a checkpoint. Here, $W_{YD} = \sqrt{\frac{2C\mu_{ind}}{p}}$. By default, we use $\mu_{ind} = 10$ years in the simulations.

Because YoungDaly is a *periodic* strategy, the size of its checkpointed segments are defined once and for all. On the contrary, NextStep adapts its checkpointing strategy to the failure history. Hence, after each failure, NextStep must recompute the size of its checkpointed segments. We take this recomputation time into account in the simulation, and conservatively add it to the recovery time[4]. To keep the recomputation time as low as possible, we introduce two optimizations to Algorithm 6. The goal is to dramatically shorten its execution time while maintaining the quality of the produced solution.

The first optimization is about the loop at Line 9. In Algorithm 6, the loop is over all possible numbers of checkpoints, ranging from a single checkpoint to one checkpoint per time quantum (i.e., W^* checkpoints). This latter solution would lead to a huge number of checkpoints. A natural conjecture is that the expected performance of NextStep would be a bell-shaped function of the number of checkpoints that are taken, first increasing and then decreasing after a threshold number has been reached. Therefore, in our implementation, we have replaced the `for` loop at Line 9 by a `while` loop that continues to look for a solution involving one additional checkpoint only if at least one of the five prior attempts leads to the best solution overall.

The second optimization is about the computation of the probability of success in Algorithm 5, Lines 4 and 13; and Algorithm 6, Line 2 and 5. This probability is the product of the individual probabilities of the processors. Hence, the execution time of this step is linear in the number of processors, while we want to consider platforms with tens of thousands of processors. Furthermore, this probability is computed many times, for many different values of i and n_c. We replace the exact computation by the following approximation. We first sort the values in τ_0^* and we retain the smallest ten and largest

[4] An alternative would be to perform this recomputation on dedicated resources, and in parallel to the recovery. We study the most costly scenario for NextStep.

ten values; then we approximate the remaining values using 100 quantiles, according to the distribution. When i and n_c vary, they add an additive term to the history, which does not change the ranking of the values. We can thus replace the exact computation by one that uses the 10 smallest and 10 largest values of the history, and the 100 quantiles with their frequency of occurrences (if there are k values for a quantile, we compute a single probability and its exponentiation rather than k probabilities that we multiply). Hence, for a pre-processing cost of $O(p \log p)$ we approximate in constant time the probability of success, since the processors that define the 120 history values remain the same up to the next failure.

Probability Distributions

We experiment with a wide range of probability distributions:

• **The Exponential distribution**, with probability density function $f(t) = \frac{e^{-t/\mu_{ind}}}{\mu_{ind}}$;

• **The Weibull distribution**, for which the probability density function has a shape parameter k, a scale parameter λ and verify $f(t;k,\lambda) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k}$. To obtain an MTBF of μ_{ind}, we chose $\lambda = \frac{\mu_{ind}}{\Gamma(1+\frac{1}{k})}$, and therefore the probability density function becomes $f(t,k) = \frac{k\Gamma(1+\frac{1}{k})}{\mu_{ind}}\left(\frac{t\Gamma(1+\frac{1}{k})}{\mu_{ind}}\right)^{k-1} e^{-\left(\frac{t\Gamma(1+\frac{1}{k})}{\mu_{ind}}\right)^k}$. In the experiments, k is varied in $\{0.5, 0.7, 1.5\}$. The first two shape values are realistic values taken from [57,138,139]; for $k < 1$, processors are more likely to fail if the processor is recent (infant mortality). The last shape value $k = 1.5$ provides an example of a distribution whose instantaneous failure rate increases with time. Note that $k = 1$ corresponds to the Exponential distribution;

• **The Gamma distribution**, with probability density function $f(t,k,\Theta) = \frac{t^{k-1}e^{-\frac{t}{\Theta}}}{\Gamma(k)\Theta^k}$, where k is the shape parameter and Θ is the scale parameter. To obtain an MTBF of μ_{ind}, we scale it using $\Theta = \frac{\mu_{ind}}{k}$ and obtain $f(t,k) = \frac{k^k t^{k-1} e^{-\frac{kt}{\mu_{ind}}}}{\Gamma(k)\mu_{ind}^k}$, where k is the shape parameter and Γ is the Gamma function. In the experiments, k is varied in $\{0.5, 0.7\}$. Note that $k = 1$ corresponds again to the exponential distribution;

• **The LogNormal distribution**, with probability density function defined as $f(t,\mu,\sigma) = \frac{1}{t\sigma\sqrt{2\pi}} e^{\left(-\frac{(\ln t-\mu)^2}{2\sigma^2}\right)}$ and with expectation $e^{\mu+\sigma^2/2}$, where μ and σ are respectively the expectation and the standard deviation of the variable's natural logarithm. We tested with two sets of (μ,σ), used in [50] and [172]: $(\mu_1 = 6.6025, \sigma_1 = 1.6206)$ and $(\mu_2 = 10.89, \sigma_2 = 1.08)$. In order to harmonize with the other probability distributions, we aim at having $\mu_{ind} = e^{\mu+\sigma^2/2} = 10$ years. To achieve this without altering the shape of the probability distribution, we fix a parameter $k = \mu/\sigma^2$, in order to express the probability density function with (μ_{ind}, k). After scaling, we obtain two sets $(\mu_{ind} = 10, k = 2.51)$ and $(\mu_{ind} = 10, k = 9.34)$ that we consider in the experiments. We retrieve the probability density function with:

$$\mu = \frac{\ln(\mu_{ind})}{1 + \frac{1}{2k}}; \qquad \sigma = \sqrt{\frac{\mu}{k}} = \sqrt{\frac{\ln(\mu_{ind})}{k + \frac{k}{2}}}.$$

Traces

We generate a failure trace for each failure distribution and for each processor. In that trace, failure IATs obey the distribution, and the last failure happens after time h, where h is the horizon of the failure trace. The horizon is set to two years ($h = 730$ days) for all

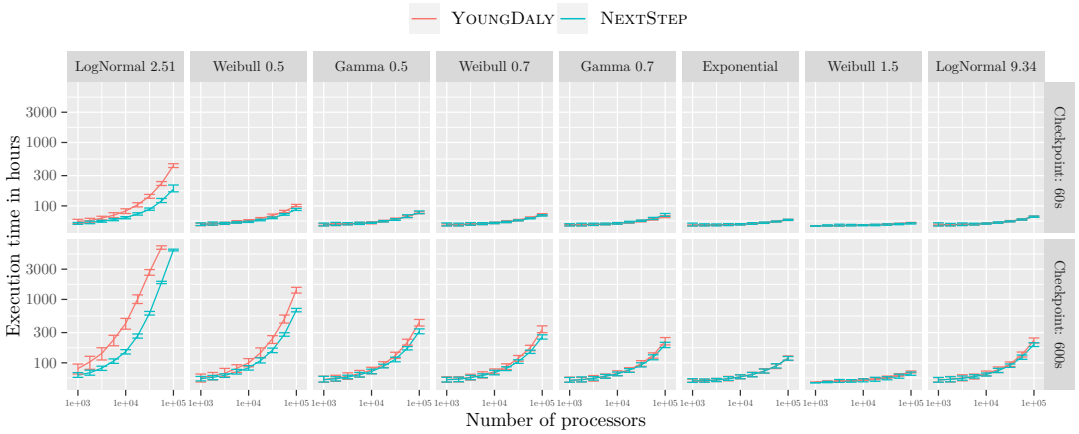


Figure 5.1: Expected performance of both heuristics under all failure distributions for a 100 days old platform with a workflow of $T_{base} = 48$ hours.

the traces. The different heuristics are then evaluated using the trace, thereby making sure that all heuristics are evaluated using the very same failure scenario. If during a simulation, a checkpointing strategy reaches time h before the completion of the application, the simulation is said to fail.

Simulation Parameters

In the experiments, we compare both checkpointing strategies under the following parameter settings:

• The number of processors p, logarithmically varied in the range 10^3 to 10^5. These values represent mid-size to large parallel platforms.

• The checkpoint/recovery/downtime $C = R = 10D$, in seconds, varied in $\{60, 600\}$. In practice, the small value of C is optimistic while the later is pessimistic; and the low value of $D = \frac{C}{10}$ assumes that spares are immediately available.

• The duration of the application T_{base}, in hours, varied in $\{1, 3, 10, 48\}$. T_{base} corresponds to the total length of the application, excluding checkpoints, if no failure occurs, when it is run on p processors (weak scaling). This corresponds to the duration range of typical HPC applications, lasting from one hour up to two days.

• The age of the platform T_{plat}, in days, varied in $\{0, 10, 30, 100, 365\}$. This is the time from which we start using the failure traces: either from their very beginning if $T_{plat} = 0$, or from a later time if $T_{plat} > 0$. The age of the platform plays an important role for non-memoryless failure distributions. At the creation of the platform, all the processors are new and without any failure history. After a failure, the processor that failed is replaced/rejuvenated, but the other processors are not and keep their history. For instance, if the processors experience infant mortality, we expect the number of failures to be much higher with $T_{plat} = 0$, when all processors are new, than after a year of service ($T_{plat} = 365$).

Evaluation Methodology

For each possible choice of parameters, we generate 50 different failure scenarios. For each failure scenario, the simulated makespan (duration of the whole execution) of both heuristics is computed. We include the time spent to compute the segment sizes of NextStep. On the plots, we report the average makespan over these 50 instances, together with the tenth

Table II: Ratio of the execution time achieved by YoungDaly to that of NextStep for the 8 failure distributions, when $T_{base} = 48$ and $T_{plat} = 100$, and when aggregating all results.

	LogNormal 2.51	Weibull 0.5	Gamma 0.5	Weibull 0.7	Gamma 0.7	Exponential	Weibull 1.5	LogNormal 9.34
T_{base}=48, T_{plat}=100	1.89 (2.02)	1.15 (1.34)	1.04 (1.17)	1.04 (1.14)	1 (1.1)	1.01 (1.06)	1.03 (1.06)	1.02 (1.11)
Aggregated	2.48 (2.26)	1.44 (1.6)	1.24 (1.43)	1.13 (1.28)	1.07 (1.21)	1.01 (1.07)	1.04 (1.07)	1.03 (1.09)

and the ninetieth percentiles, as a function of the number of processors. The YoungDaly heuristic is shown in red, and NextStep in blue. In all figures, the y-axis is the makespan in hours, and the x-axis corresponds in most cases to the number of processors; both axes are in log-scale.

In the tables, we report the relative performance of YoungDaly and NextStep. More precisely, for each failure scenario, we compute the ratio of the makespan achieved by YoungDaly divided by that of NextStep. Hence, NextStep achieves a better makespan when the ratio is greater than 1; the larger the ratio, the higher the benefit of using NextStep. To produce meaningful statistics on these ratios, we compute and report their geometric mean and geometric standard deviation (in parentheses). For a few configurations, YoungDaly does not succeed to complete the application before the trace horizon. For these cases, in order to be able to compute statistics, we take for the execution time of YoungDaly a lower bound, namely the time at which the execution was stopped: $h - T_{plat}$. We checked that using this lower-bound or computing the statistics while just discarding these configurations leads to almost identical results (differences below 1%). The simulation code for all experiments is publicly available at `http://perso.ens-lyon.fr/frederic.vivien/resilience/non-memoryless-checkpoint`.

5.6.2 Results

We first compare the behavior of both checkpointing heuristics with the different probability distributions on a particular set of parameters, before studying the impact of the different parameters.

Comparison of Probability Distributions

Figure 5.1 compares the two heuristics for the different failure distributions, with a checkpoint length of one or ten minutes, where the application length is 48 hours and the platform is 100 days old. In this case, although the platform is not new, we see that the NextStep heuristic is performing either better than or similarly to YoungDaly. Moreover, the difference tends to be more important when the checkpoint length is higher (bottom graphs). Recall that lower is better, since we plot execution times.

Although the MTBF of any individual processor is the same for all failure distributions ($\mu_{ind} = 10$ years), the shape of these distributions significantly impacts the number of failures that occur during the processing of the application, as well as the distribution of the failures. For instance, if the processors tend to have infant mortality (which corresponds to distributions on the left of the figure), and if the platform is not very old, then applications may actually experience more failures than expected. This is the case for the LogNormal distribution with $k = 2.51$ or Weibull with $k = 0.5$ in Figure 5.1. This explains the higher execution times of YoungDaly for both heuristics.

Furthermore, YoungDaly does not checkpoint often enough, as it considers the global long term MTBF of the platform instead of its actual instantaneous failure rate. This

Table III: Ratio of the execution time achieved by YoungDaly to that of NextStep for the 8 failure distributions for the different platform sizes and when averaging over all the other parameters.

Platform size	LogNormal 2.51	Weibull 0.5	Gamma 0.5	Weibull 0.7	Gamma 0.7	Exponential	Weibull 1.5	LogNormal 9.34
1000	1.34 (1.6)	1.14 (1.33)	1.08 (1.22)	1.03 (1.11)	1.01 (1.08)	1 (1.04)	1.01 (1.04)	1.01 (1.04)
1778	1.53 (1.82)	1.18 (1.41)	1.11 (1.3)	1.03 (1.12)	1.02 (1.1)	1 (1.04)	1.01 (1.04)	1.01 (1.04)
3162	1.88 (2.07)	1.26 (1.49)	1.16 (1.37)	1.05 (1.16)	1.02 (1.13)	1.01 (1.04)	1.02 (1.04)	1.01 (1.05)
5623	2.22 (2.28)	1.33 (1.54)	1.19 (1.38)	1.07 (1.19)	1.04 (1.14)	1.01 (1.06)	1.03 (1.04)	1.02 (1.06)
10000	2.72 (2.33)	1.38 (1.58)	1.23 (1.42)	1.09 (1.24)	1.06 (1.18)	1.01 (1.05)	1.03 (1.05)	1.02 (1.06)
17783	3.2 (2.33)	1.51 (1.66)	1.26 (1.44)	1.14 (1.29)	1.08 (1.22)	1.01 (1.07)	1.04 (1.06)	1.03 (1.07)
31623	3.74 (2.18)	1.72 (1.72)	1.36 (1.5)	1.22 (1.36)	1.12 (1.27)	1.02 (1.08)	1.07 (1.08)	1.05 (1.12)
56234	3.65 (2.02)	1.76 (1.63)	1.37 (1.5)	1.24 (1.35)	1.14 (1.3)	1.01 (1.1)	1.08 (1.09)	1.05 (1.13)
100000	3.5 (1.92)	1.85 (1.55)	1.41 (1.48)	1.33 (1.41)	1.15 (1.32)	1.01 (1.09)	1.08 (1.1)	1.07 (1.16)

Table IV: Ratio of the execution time achieved by YoungDaly to that of NextStep for the 8 failure distributions as a function of platform age, with $p = 56234$ and $T_{base} = 48$ and when averaging over the two checkpoint sizes. The last column provides an average over all distributions.

Platform age	LogNormal 2.51	Weibull 0.5	Gamma 0.5	Weibull 0.7	Gamma 0.7	Exponential	Weibull 1.5	LogNormal 9.34	**Average**
0	4.17 (2.06)	2.33 (1.48)	1.85 (1.44)	1.42 (1.38)	1.28 (1.32)	1.03 (1.08)	1.08 (1.07)	1.08 (1.07)	**1.58** (1.78)
10	3.27 (2.26)	1.61 (1.66)	1.29 (1.46)	1.13 (1.29)	1.06 (1.2)	1.01 (1.06)	1.04 (1.06)	1.02 (1.06)	**1.31** (1.71)
30	2.57 (2.17)	1.36 (1.55)	1.15 (1.32)	1.08 (1.21)	1.03 (1.16)	1 (1.06)	1.03 (1.06)	1 (1.06)	**1.21** (1.58)
100	1.89 (2.02)	1.15 (1.34)	1.04 (1.17)	1.04 (1.14)	1 (1.1)	1.01 (1.06)	1.03 (1.06)	1.02 (1.11)	**1.12** (1.42)
365	1.42 (1.72)	1.05 (1.17)	1.01 (1.1)	1.02 (1.11)	1 (1.08)	1 (1.06)	1.01 (1.07)	1.03 (1.13)	**1.06** (1.27)

is because YoungDaly does not take the failure history into account. On the contrary, NextStep does take that history into account. Therefore, it correctly estimates the instantaneous failure rate. This results in a makespan that can be up to two times lower.

There are some distributions for which processors tend to be more robust at the beginning because of their young age (distributions on the right of the figure). In this case, when the platform is rather young, the number of failures is lower than what would be expected regarding the MTBF of the platform. A good example is the Weibull distribution with $k = 1.5$ in Figure 5.1. In that case, the actual instantaneous failure rate of the platform is lower than expected, YoungDaly tends to over-checkpoint because it does not take into account this actual failure rate, whereas NextStep adapts its checkpointing strategy according to this history, showing once again its versatility. Yet this time, the difference between heuristics is low, because the overall checkpointing cost remains small in both cases.

Finally, if the platform actual instantaneous failure rate is in accordance with the expected MTBF, as is the case for an Exponential distribution, YoungDaly is optimal. We check that the performance of NextStep and YoungDaly are similar in this setting.

Altogether, these results show that NextStep always adapts to the actual instantaneous failure rate, because it accounts for the failure history of processors. Its versatility makes it a better strategy in all the cases: Table II summarizes the results, reporting the ratio of the execution time achieved by YoungDaly to that of NextStep (geometric average, geometric standard deviation). We point out that the difference is more significant for the realistic probability distributions that have been advocated in the literature: namely Weibull with a shape parameter smaller than one [57, 138, 139], and LogNormal [50, 172].

The previous study was for long applications lasting 48 hours and $T_{plat} = 100$. We also present the aggregated results over all application lengths and platform ages in Table II. We observe that NextStep achieves even larger gains. For instance, for LogNormal 2.51, the average ratio becomes 2.48, instead of 1.89 for the scenario with $T_{plat} = 100$.

Table V: Ratio of the execution time achieved by YoungDaly to that of NextStep for the 8 failure distributions for the two checkpoint durations (in seconds) when averaging over all the other parameters.

Checkpoint duration	LogNormal 2.51		Weibull 0.5		Gamma 0.5		Weibull 0.7		Gamma 0.7		Exponential		Weibull 1.5		LogNormal 9.34	
60	2.18	(2.27)	1.33	(1.52)	1.17	(1.37)	1.08	(1.2)	1.03	(1.14)	1	(1.03)	1.02	(1.04)	1.01	(1.04)
600	2.83	(2.2)	1.56	(1.66)	1.3	(1.47)	1.18	(1.34)	1.11	(1.26)	1.02	(1.09)	1.06	(1.08)	1.05	(1.12)

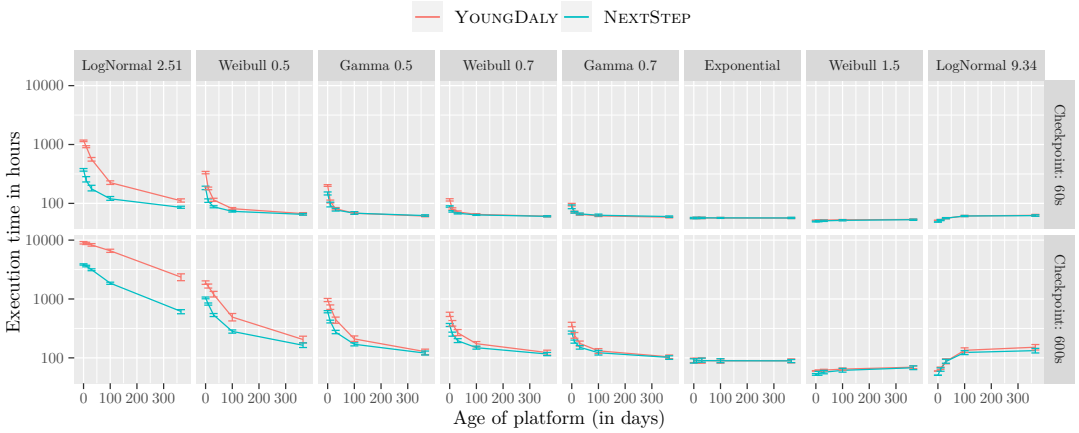


Figure 5.2: Expected performance of both heuristics under all failure distributions, with $p = 56234$ and $T_{base} = 48$ hours.

Impact of the Different Parameters

Impact of the number of processors. It can be observed on Figure 5.1: the more processors, the more failures, and the larger the makespan for both heuristics, as one could have foretold. In most settings, the performance of YoungDaly worsens relatively to that of NextStep when the number of processors increases (recall that the y-axis is in log-scale). Again, this can be explained as follows: the difference between the estimated failure rate and the instantaneous failure rate increases with the number of processors; hence, worse results for YoungDaly. On the contrary, NextStep adapts to the instantaneous failure rate. Table III provides a comprehensive summary of results for each platform size, averaging over all other parameters. The table confirms this observation.

Impact of the age of the platform. The age of the platform has a great impact on the performance of both heuristics, because the instantaneous failure rate of the platform highly depends on it. When processors have a high infant mortality, a younger platform leads to more errors and thus to a higher makespan for both heuristics. This can be observed in Figure 5.2, especially on the leftmost graphs. On this figure, the x-axis is now the age of the platform (in a linear scale). The number of processors is fixed to $p = 56234$ and the application execution time is $T_{base} = 48$ hours.

For all distributions to the left of the Exponential, the newer the platform, the higher the difference between the heuristics. Indeed, for younger platforms, processors are more likely to fail due to infant mortality; and the older the platform, the more the instantaneous failure rate resembles an Exponential. The same observation can be made for Weibull 1.5, although in this case, this is due to low infant mortality. Indeed, with this distribution, less failures occur for younger platforms.

LogNormal 9.34 behaves slightly differently. For this distribution, once again, YoungDaly does not adapt to the instantaneous failure rate: either it underestimates the

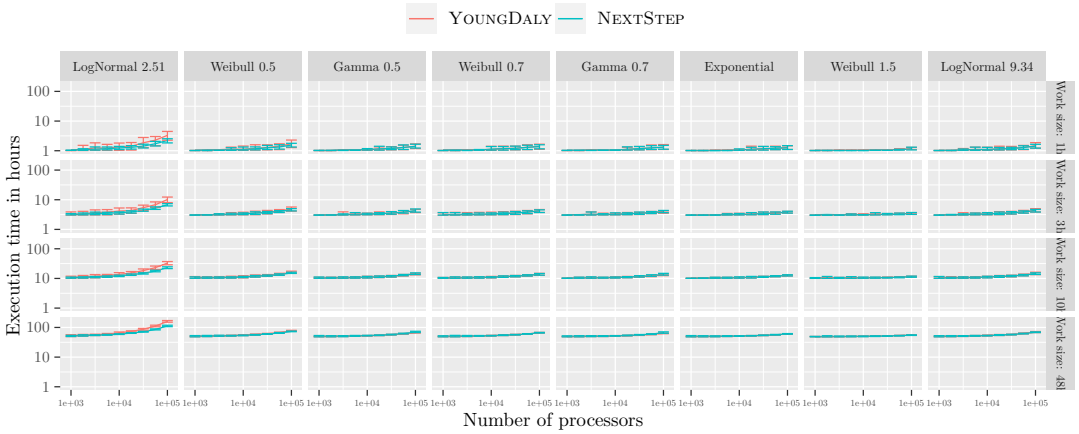


Figure 5.3: Expected performance of both heuristics under all failure distributions on a 365 day old platform, with $C = 60$ seconds.

instantaneous failure rate of a new platform and does not checkpoint enough, or it overestimates the instantaneous failure rate of an old platform and checkpoints too much. On the contrary, NextStep adjusts the checkpointing strategy for both cases. For intermediate platform ages, both heuristics have close performance because this is where the instantaneous failure rate is closest to what is expected ($\frac{\mu_{ind}}{p}$) by YoungDaly. Nevertheless, the variance is different from that of the Exponential distribution, and NextStep achieves slightly better performance.

Table IV summarizes these results. Most gains are obtained for young platforms. NextStep always achieves a performance at least similar to YoungDaly, and much better in many cases.

Impact of the checkpoint time. As expected, the larger the checkpoint cost, the larger the execution time for both heuristics, as shown in Figure 5.1. Having a larger checkpoint cost exacerbates the differences between both heuristics. Indeed, when checkpoints cost more, both heuristics execute fewer checkpoints and thus lose more time at each failure. In the end, this increases the performance loss due to a bad checkpointing strategy. Table V summarizes the results for the two checkpoint costs (one minute or ten minutes).

Impact of the application length. Again, the larger the application length, the larger the execution time for both heuristics, as shown in Figure 5.3. Moreover, the error bars are much wider for a small workload, because having larger applications will smooth the impact of each individual failure. Table VI summarizes results for the four application lengths by aggregating all results. Overall, more gain can be achieved with smaller application lengths. Indeed, relative to the lengths of applications, checkpoints are more expensive for small applications. This conclusion is similar to that on the impact of the cost of checkpoints. This phenomenon can be observed by comparing Tables V and VI, where the impact of increasing the checkpoint cost is similar to the impact of decreasing the application length.

5.7 Conclusion

In this chapter, we have investigated checkpointing strategies to protect parallel applications from non-memoryless failures. Indeed, the optimal strategy has been well studied when failure IATs obey an Exponential distribution, but it is not well understood for non-memoryless failure distributions. We have designed a general strategy, NextStep, which

Table VI: Ratio of the execution time achieved by YoungDaly to that of NextStep for the 8 failure distributions for the different application lengths (in hours) and when averaging over all the other parameters.

application length	LogNormal 2.51		Weibull 0.5		Gamma 0.5		Weibull 0.7		Gamma 0.7		Exponential		Weibull 1.5		LogNormal 9.34	
1	2.36	(2.84)	1.42	(1.81)	1.25	(1.58)	1.12	(1.39)	1.07	(1.29)	1.02	(1.09)	1.05	(1.08)	1.04	(1.12)
3	2.83	(2.47)	1.52	(1.71)	1.29	(1.49)	1.15	(1.32)	1.09	(1.25)	1.01	(1.07)	1.04	(1.08)	1.03	(1.1)
10	2.63	(2)	1.47	(1.51)	1.25	(1.35)	1.14	(1.23)	1.07	(1.17)	1	(1.05)	1.04	(1.06)	1.03	(1.08)
48	2.16	(1.6)	1.35	(1.32)	1.16	(1.21)	1.11	(1.15)	1.05	(1.11)	0.997	(1.03)	1.03	(1.05)	1.02	(1.06)

maximizes the expected efficiency until the next failure. While it may not be optimal because of side-effects towards the end of the application, we proved that this strategy is asymptotically optimal for very long applications.

Instead of maximizing the expected efficiency until the next failure, traditional solutions consist in checkpointing periodically according to the platform MTBF (YoungDaly strategy). Our extensive simulation results show that this strategy works well for Exponential distributions, but not for the other distributions, because it either underestimates or overestimates the actual instantaneous failure rate. On the contrary, NextStep is always at least as good as YoungDaly for any failure distribution, and significantly outperforms it in many cases. Overall, our study demonstrates the interest of always using NextStep instead of YoungDaly.

In particular, the difference between NextStep and YoungDaly is very important for distributions whose infant mortality of the distribution is high, e.g. LogNormal 2.51 or Weibull 0.5. The latter distributions have been advocated to model failures on real-life platforms [57, 79, 138, 139, 150, 151], which further evidences the impact and significance of NextStep.

Future work will focus on checkpointing strategies for workflows composed of parallel tasks with dependencies, instead of single parallel applications as in this study. The criticality of some tasks in the workflow may lead to checkpoint them more often than prescribed by the NextStep strategy tuned for a given non-memoryless failure distribution.

Chapter 6

Checkpointing Workflows à la Young/Daly Is Not Good Enough

In Chapter 5, we have studied checkpointing strategies for non-memoryless failure distributions with a tightly-coupled preemptible application, and showed the limits of the Young/Daly formula. In this chapter, we assume memoryless failure distributions, and show that Young/Daly is again perfectible when workflows composed of multiple tasks execute on a parallel platform, and we propose novel checkpointing strategies for this case. Again, the objective is to minimize the expectation of the total execution time. For a single task, the Young/Daly formula provides the optimal checkpointing period. However, when many tasks execute simultaneously, the risk that one of them is severely delayed increases with the number of tasks. To mitigate this risk, a possibility is to checkpoint each task more often than with the Young/Daly strategy. But is it worth slowing each task down with extra checkpoints? Does the extra checkpointing make a difference globally? This chapter answers these questions. On the theoretical side, we prove several negative results for keeping the Young/Daly period when many tasks execute concurrently, and we design novel checkpointing strategies that guarantee an efficient execution with high probability. On the practical side, we report comprehensive experiments that demonstrate the need to go beyond the Young/Daly period and to checkpoint more often, for a wide range of application/platform settings. This chapter corresponds to Publication [J2] (see Chapter 9); some proofs that are omitted here for conciseness are available in the research report [R2].

6.1 Introduction

Checkpointing is the standard technique to protect applications running on HPC (High Performance Computing) platforms. Every day, the platform will experience a few fail-stop errors (or failures; we use both terms indifferently). After each failure, the application executing on the faulty processor (and likely on many other processors for a large parallel application) is interrupted and must be restarted. Without checkpointing, all the work executed for the application is lost. With checkpointing, the execution can resume from the last checkpoint, after some downtime (enroll a spare to replace the faulty processor) and a recovery (read the checkpoint).

Consider an application, composed of a unique task, executing on a platform whose nodes are subject to fail-stop errors. Say the application executes for $T_{base} = 10$ hours, can checkpoint in $C = 6$ minutes, and experiences failures whose inter-arrival times follow an Exponential distribution with mean $\mu = 4$ hours. This means that a failure strikes the

P	Total number of processors
n	Number of tasks
$\mu_{ind} = \frac{1}{\lambda}$	Individual processor's MTBF
C	Checkpoint time
R	Recovery time
D	Downtime
T_{base}	Task duration without failures
N_{opt}	Number of segments of a task with LAMBERT strategy
N_{ME}	Number of segments of a task with MINEXP strategy
W_{ME}	Segment length of a task with MINEXP strategy
Δ	max. number of tasks processed concurrently by the failure-free schedule

Table I: Summary of main notations for Chapter 6

application every 4 hours in expectation (see Section 6.2.1 for details). Assume a short downtime $D = 1$ minute, and a recovery time $R = C$. How frequently should the task be checkpointed so that its expected execution time $\mathbb{E}(T_{1\text{-}task})$ is minimized? There is a well-known trade-off: taking too many checkpoints leads to a high overhead, especially when there are few failures, while taking too few checkpoints leads to a large re-execution time after each failure. Here is an illustration of this trade-off:

- If we take no checkpoint at all, then the expected execution time is $\mathbb{E}(T_{1\text{-}task}) \approx 46$ hours (see Equation (5.1) in Section 6.2.5 to derive this value);
- If we take a checkpoint at the end of the execution, e.g., to save final results on stable storage[1], then $\mathbb{E}(T_{1\text{-}task})$ increases by about 76 minutes, which is surprising given the short checkpoint time; but keep in mind that if a failure strikes during the checkpoint, which happens with a low probability of 2.5%, then the 10 hours of execution are wasted;
- If we checkpoint every hour (an application-agnostic approach that has been implemented for several HPC platforms [80]), then we have 10 equal-length segments, each of duration of 1 hour and followed by a checkpoint. We obtain $\mathbb{E}(T_{1\text{-}task}) \approx 13$ hours. Checkpointing every hour brings a huge benefit!
- Finally, if we checkpoint every 20 minutes, then we obtain $\mathbb{E}(T_{1\text{-}task}) \approx 14$ hours. Checkpointing too frequently becomes an overkill.

As seen in the previous chapter, the optimal checkpointing period is given by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$ [47, 173], where μ is the application MTBF (Mean Time Between Failures, and C the checkpoint duration. In the example above with $\mu = 4$ hours and $C = 6$ minutes, we obtain $W_{YD} \approx 54$ minutes. This value would be the optimal checkpointing period for a task of infinite length. For a task of length $T_{base} = 10$ hours, the optimal solution (see Section 6.2.5) is to use either $\max(1, \lfloor \frac{T_{base}}{W_{YD}} \rfloor) = 11$ or $\lceil \frac{T_{base}}{W_{YD}} \rceil = 12$ equal-length segments, whichever leads to the smaller $\mathbb{E}(T_{1\text{-}task})$. We find that the best value is $\mathbb{E}(T_{1\text{-}task}) \approx 13$ hours with 12 segments of length 50 minutes. The best value is smaller by only 1 minute than the value with 11 segments, and by only 3 minutes than the value with 10 segments, which shows the robustness of the approach.

We now move to a more complicated example and assume that 300 independent applications have been launched concurrently on the platform. These 300 applications

[1]We make this assumption throughout the chapter for simplicity. We can easily extend the analysis to the case where no checkpoint is taken at the end of the execution of a task. Changes are minimal and results are quite similar. The details are omitted in this thesis but can be found in [R2, Appendix A]

are identical to the application above: each has a unique task of length $T_{base} = 10$ hours, checkpoint duration $C = 6$ minutes, recovery time $R = C$, and the downtime is $D = 1$ minute. For the example to be more realistic in terms of failure rate, we assume that each application executes with $p = 30$ processors. Hence, the platform has at least $m = 9,000$ processors. Each processor is subject to failures following an Exponential distribution $Exp(\frac{1}{\mu_{ind}})$, where μ_{ind} is the individual processor's MTBF. Since each task executes on $p = 30$ processors, its MTBF is $\mu = \frac{\mu_{ind}}{p}$. In other words, the MTBF of a task is inversely proportional to the number of processors enrolled, which is intuitive in terms of failure frequency (see [80] for a formal proof). We now assume that each task has 0.5% chances to fail during execution; this setting corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$, i.e., $\mu_{ind} = 59,850$ hours (or 6.8 years). This is in accordance with MTBFs typically observed on large-scale platforms, which range from a few years to a few dozens of years [36]. For each task, the Young/Daly period is $W_{YD} = \sqrt{2\frac{\mu_{ind}}{p}C} \approx 20$ hours, and the expected execution time of a single task $\mathbb{E}(T_{1\text{-}task})$ is minimized either when no checkpoint is taken or if a single checkpoint is taken at the end of the execution (see Section 6.2.6). Recall that, as stated above, we assume that we always take a checkpoint at the end of the execution of a task. Then, we derive that $\mathbb{E}(T_{1\text{-}task}) \approx 10.4$ with a single checkpoint taken at the end of each task (see Section 6.2.6 for details of this computation).

Is it safe to checkpoint each task individually à la Young/Daly? The problem comes from the fact that the expectation $\mathbb{E}(T_{all\text{-}tasks})$ of the maximum execution time over all tasks, i.e., the expectation of the total time required to complete all tasks, is far larger than the maximum of the expectations (which in the example have all the same value $\mathbb{E}(T_{1\text{-}task})$). When a single checkpoint is taken at the end of each task, we compute that $\mathbb{E}(T_{all\text{-}tasks}) > 14$, while adding four intermediate checkpoints to each task reduces it down to $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$ (see Section 6.2.6 for details of the computation of both numbers). Intuitively, this is because adding these intermediate checkpoints greatly reduces the chance of re-executing any single task from scratch when it is struck by a failure, and the probability of having at least one failed task increases with the number of tasks. Of course, there is a penalty from the user's point of view: Adding four checkpoints to each task augments their length by 24 minutes, while the majority of them will not be struck by a failure. In other words, users may feel that their response time has been unduly increased, and state that it is not worth to add these extra checkpoints.

Going one step further, consider now a single application whose dependence graph is a simple fork-join graph, made of 302 tasks: an entry task, 300 parallel tasks identical to the tasks above (each task runs on $p = 30$ processors for $T_{base} = 10$ hours, and is checkpointed in $C = 6$ minutes) and an exit task. Such applications are typical of HPC applications that explore a wide range of parameters or launch subproblems in parallel. Now, the extra checkpoints make full sense, because the exit task cannot start before the last parallel task has completed. The expectation of the total execution time is $\mathbb{E}(T_{total}) = \mathbb{E}(T_{entry}) + \mathbb{E}(T_{all\text{-}tasks}) + \mathbb{E}(T_{exit})$, where $\mathbb{E}(T_{entry})$ and $\mathbb{E}(T_{exit})$ are the expected durations of the entry and exit tasks, and $\mathbb{E}(T_{total})$ is minimized when $\mathbb{E}(T_{all\text{-}tasks})$ is minimized. By diminishing $\mathbb{E}(T_{all\text{-}tasks})$, we save 1.25 hour, or 75 minutes (and in fact much more than that, because the lower and upper bounds for $\mathbb{E}(T_{all\text{-}tasks})$ are loosely computed).

This last example shows that the optimal execution of large workflows on failure-prone platforms requires to checkpoint each workflow task more frequently than prescribed by the Young/Daly formula. The main focus of this chapter is to explore various checkpointing strategies, and our main contributions are the following:

- We provide approximation bounds for the performance of MinExp, a strategy à la

Young/Daly that minimizes the expected execution time of each task, and for a novel strategy CheckMore that performs more checkpoints than MinExp.

- Both bounds apply to workflows of arbitrary shape, and whose tasks can be either rigid or moldable. In addition, we exhibit an example where the bounds are tight and where CheckMore can be an order of magnitude better than MinExp.
- The novel CheckMore strategy comes in two flavors, one that tunes the number of checkpoints as a function of the degree of parallelism in the failure-free schedule, and a simpler one that does not require any knowledge of the failure-free schedule, beyond a priority list to decide in which order to start executing the tasks.
- We report comprehensive simulations results based on WorkflowHub testbeds [62], which demonstrate the significant gain brought by CheckMore over MinExp for almost all testbeds.

The chapter is organized as follows. We first describe the model in Section 6.2. We assess the performance of MinExp in Section 6.3; performance bounds are proven both for independent tasks and for general workflows. Section 6.4 presents the novel strategy CheckMore that checkpoints workflow tasks more often than MinExp, and analyzes its theoretical performance. The experimental evaluation in Section 6.5 presents extensive simulation results comparing both strategies. Finally, we discuss related work in Section 6.6, and conclude in Section 6.7.

6.2 Model and Background

In this section, we first detail the platform and application models, and describe how to practically deploy a workflow with checkpointed jobs. Then, we discuss the objective function before providing background on the optimal checkpointing period for preemptible tasks, and getting back to the example of the introduction. Key notations are summarized in Table I.

6.2.1 Platform

We consider a large parallel platform with P identical processors, or nodes. These nodes are subject to fail-stop errors, or failures. A failure interrupts the execution of the node and provokes the loss of its whole memory. There are many causes of failures, including power outages or network errors, and they cause the node to stall or crash [56, 139]. Consider a parallel application running on several nodes: when one of these nodes is struck by a failure, the state of the application is lost, and execution must restart from scratch, unless a fault-tolerance mechanism has been deployed.

The classical technique to deal with failures makes use of a checkpoint-restart mechanism: the state of the application is periodically checkpointed, i.e., all participating nodes take a checkpoint simultaneously. This is the standard coordinated checkpointing protocol, which is routinely used on large-scale platforms [39], where each node writes its share of the application data to stable storage (checkpoint of duration C). When a failure occurs, the platform is unavailable during a downtime D, which is the time to enroll a spare processor that will replace the faulty processor [47, 80]. Then, all application nodes (including the spare) recover from the last valid checkpoint in a coordinated manner, reading the checkpoint file from stable storage (recovery of duration R). Finally, the execution is resumed from that point on, rather than starting again from scratch. Note that failures can strike during checkpoint and recovery, but not during downtime (otherwise we can include the downtime in the recovery time).

Throughout the chapter, we add a final checkpoint at the end of each application task, to write final outputs to stable storage. Symmetrically, we add an initial recovery when re-executing the first checkpointed segment of a task (to read inputs from stable storage) if it has been struck by a failure before completing the checkpoint. These assumptions are done here for simplicity but have a negligible impact. Some more details can be found in [R2, Appendix A], and they are omitted from this thesis for conciseness.

We assume that each node experiences failures whose inter-arrival times follow an Exponential distribution $Exp(\lambda)$ of parameter $\lambda > 0$, whose PDF (Probability Density Function) is $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. The individual MTBF of each node is $\mu_{ind} = \frac{1}{\lambda}$. Even if each node has an MTBF of several years, large-scale parallel platforms are composed of so many nodes that they will experience several failures per day [36, 61]. Hence, a parallel application using a significant fraction of the platform will typically experience a failure every few hours.

6.2.2 Application

We focus on HPC applications expressed as workflow graphs, such as those available in WorkflowHub [62] (formerly Pegasus [154]). The shape of the task graph is arbitrary, and the tasks can be parallel. We further assume that all tasks are preemptible, i.e., that we can take a checkpoint at any instant.

For the theoretical analysis, we use workflows whose tasks can be rigid or moldable parallel tasks. A moldable task can be executed on an arbitrary number of processors, and its execution time depends on the number of processors allotted to it. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (rigid tasks) and a variable dynamic allocation (malleable tasks) [59]. Scheduling rigid or moldable workflows is a difficult NP-hard problem (see the related work in Section 6.6) . We take as input a failure-free schedule for the workflow and transform it by adding checkpoints as follows. The failure-free schedule provides an ordered list of tasks, sorted by non-decreasing starting times. Our failure-aware algorithms are list schedules that greedily process the tasks (augmented with checkpoints) in this order: if task T is number i in the original failure-free schedule, then T is scheduled after the $i-1$ first tasks in the failure-aware schedule, and no other task can start before T does. Hence, the processors allocated to T in the failure-aware schedule may differ from those allocated in the failure-free schedule. Enforcing the same ordering of execution of the tasks may be sub-optimal, but it is the key to guarantee approximation ratios for the total execution time.

For the experiments, we restrict to workflows with uni-processor tasks, in accordance with the characteristics of the workflow benchmarks from WorkflowHub.

6.2.3 Implementation in a Cluster Environment

This section briefly describes two approaches to deploy a workflow with checkpointed jobs in a cluster environment.

The first approach is to use the job scheduler LSF [87] and to submit a set of jobs with their dependencies: there are as many jobs as tasks in the workflow, and these jobs are declared *checkpointable*. The system will relaunch a job after it is hit by a failure, from the last checkpoint on and until success (see the 'job failover' section in [87]). If the failed job was using j processors, then it releases $j-1$ surviving processors right after the failure; if there is at least one other processor available, the job can be rescheduled right away (jobs usually get high priority when they are rescheduled after a failure). Otherwise, the failed

job will have to wait and this waiting time, a.k.a. the re-submission time, is dependent on the platform scheduling policy and on the availability of nodes.

A second approach is to submit a single job with $p + q$ processors, where p processors represent the allotment for the whole workflow and q processors are spare. The job uses a master process that spans the workflow tasks and controls how their execution progresses; the tasks are checkpointed using a standard software such as VeloC [37]. The spare nodes are mutualized across the tasks either by using a fault tolerant MPI library like ULFM [28, 58], or by having the master process launch each task as independent MPI applications spanning on subgroups of the reservation, and re-launching them from their last checkpoint on the surviving nodes and the spare nodes if some task is subject to failure.

In the first approach, the downtime would be non-constant, because it corresponds to the re-submission time, while in the second approach with spares, the downtime can be approximated as a constant. Regardless, all the results of this chapter are taken in expectation, and they extend to using an average value of the downtime whenever a fixed value is not appropriate.

Finally, we stress that this work is agnostic of system management policies and does not modify any parameter specified by the user for the job allocations; we simply increase the checkpoint frequency when needed, which results in shorter execution time and better processor utilization for the workflow.

6.2.4 Objective Function

Given a workflow composed of a set of tasks, where each task executes on a given number of processors, the objective function is to minimize the expected makespan of the workflow, i.e., the expected total execution time to complete all tasks. We aim at determining the best checkpointing strategy for the tasks that compose the workflow. This is the only parameter that we modify in the execution: we keep the number of processors specified by the user, and we even keep the order of the tasks as given by the user schedule. The replacement of failed nodes, or the resubmission of failed tasks, is decided by the system and does not depend upon the checkpointing policy, either à la Young/Daly, or one of our new strategies.

As a result, minimizing the expected makespan of the workflow also maximizes processor utilization of the platform, because the processors reserved by the user will be released earlier on, and with no additional cost for the rest of the platform.

In the analysis of the checkpointing strategies, we focus on bounding the *ratio*, which is defined as the expected makespan of the workflow (i.e., the expected total execution time) divided by the makespan in the failure-free execution (no checkpoints nor failures), given a user-specified schedule. Hence, the ratio shows the overhead induced by failures and the checkpointing strategy: the closer to one, the better.

6.2.5 MinExp Checkpointing strategy

Following the results of Section 5.3.2, we define the MinExp strategy as follows:

Definition 3. *The MinExp checkpointing strategy partitions a parallel task of length T_{base}, with p processors and checkpoint time C, into $N_{ME} = \left\lceil \frac{T_{base}}{W_{YD}} \right\rceil$ equal-length segments, each followed by a checkpoint, where $W_{YD} = \sqrt{\frac{2C}{p\lambda}} = \sqrt{\frac{2\mu_{ind}C}{p}}$. Each segment is of length $W_{ME} = \frac{T_{base}}{N_{ME}}$.*

The experimental results in Section 6.5 show that using N_{ME}, whose value is based upon the Young/Daly formula, leads to almost the same results as when using N_{opt}, whose value

is based on the Lambert function. Because we are assuming the failure probability function follow the exponential law MinExp is near-optimal in expectation for each individual task.

6.2.6 Back to the Example

In the introduction, we used the example of 300 identical tasks, each with $T_{base} = 10$ hours, $p = 30$, and $C = 6$ minutes. We also had $D = 1$ minute and $R = C$. We assume that each task has 0.5% chances to fail during execution, which corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$. This equality leads to $\mu_{ind} = 59,850$ hours. We derive $W_{YD} = \sqrt{2\mu_{ind}C/p} \approx 20$ hours, hence $N_{ME} = 1$. With a single segment, we then compute the optimal expected execution time $\mathbb{E}(T_{1\text{-}task})$ for each task as:

$$\mathbb{E}(T_{1\text{-}task}) = \left(\frac{\mu_{ind}}{p} + D\right) e^{\frac{pR}{\mu_{ind}}} \left(e^{\frac{p}{\mu_{ind}}(T_{base}+C)} - 1\right) \approx 10.4.$$

With 300 tasks executing concurrently, we compute that the expectation of the total time required to complete all tasks is at least $\mathbb{E}(T_{all\text{-}tasks}) > 14$, hence the ratio is $\frac{14}{10} = 1.4$. Indeed, there is no failure at all with probability $\left(e^{-\frac{p(T_{base}+C)}{\mu_{ind}}}\right)^{300} < 0.23$, and in this case the execution time is $T_{base} + C = 10.1$. The other case, happening with a probability larger than 0.77, is when at least one failure occurs in the process, and we will bound its expected execution time if exactly one failure occurs, which is clearly lower than the actual expected execution time. To that end, we compute the expected time lost before the failure occurs when attempting to successfully execute for $T = T_{base}+C$ hours: $\mathbb{E}(T_{lost}(T)) = \int_0^\infty x\mathbb{P}(X = x|X < T)dx = \frac{1}{\mathbb{P}(X<T)} \int_0^T xp\lambda e^{-p\lambda x}dx$, with $\mathbb{P}(X < T) = 1 - e^{-p\lambda T}$. Integrating by parts, we derive that:

$$\mathbb{E}(T_{lost}(T)) = \frac{1}{p\lambda} - \frac{T}{e^{p\lambda T} - 1}. \tag{6.1}$$

In the example, we have $T = T_{base} + C = 10.1$, $p = 30$, and $\lambda = \frac{1}{\mu_{ind}} = \frac{-\ln(0.995)}{pT_{base}}$. Thus, if a failure strikes one of the tasks, the expected time lost is higher than 5.045 hours. After that, we also have to wait $D > 0.016$ hour of downtime and recover for a duration of $R = 0.1$ hour. Overall, the expected execution time satisfies $\mathbb{E}(T_{all\text{-}tasks}) \geq 10.1 + 0.77 \times (\mathbb{E}(T_{lost}(T)) + R + D) > 10.1 + 0.77 \times 5.161 > 14$. Note that this lower bound is far from tight.

When adding four intermediate checkpoints to each task, we obtain $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$. Indeed, the tasks are now slightly longer (10.5 hours without failure), and they fail with probability $1 - e^{-\frac{30\times10.5}{59850}} < 0.006$. Let M_f denote the maximum number of failures of any tasks. Clearly, we have $\mathbb{P}\{M_f \geq k\} \leq 300 \times 0.006^k$. The worst-case scenario for each failure is when it happens just before the end of a checkpoint, and in that case we loose at most $2 + 0.1 + 0.1 + 0.017 < 2.22$ for each failure (the length of a segment, the checkpoint time, the recovery time and the downtime). Thus, $\mathbb{E}(T_{all\text{-}tasks}) < 10.5 + 2.22 \sum_{k\geq1} \mathbb{P}\{M_f \geq k\} < 10.5 + 2.22 + 2.22 \times 300 \times \sum_{k\geq2} 0.006^k < 12.75$, hence a ratio lower than 1.275, to compare with 1.4 with the MinExp strategy. Note that this upper bound is far from tight. This example shows that the optimal checkpointing strategy should not only be based upon the task profiles, but also upon the number of other tasks that are executing concurrently.

6.3 Young/Daly for Workflows: the MinExp Strategy

In this section, we prove performance bounds for the MinExp checkpointing strategy, which adds N_{ME} checkpoints to each task, thereby minimizing the expected execution time

for each task. We start in Section 6.3.1 with independent tasks, first identical and then arbitrary, that can be executed concurrently (think of a shelf of tasks). Next, we move to general workflows in Section 6.3.2.

6.3.1 MinExp for Independent Tasks

We start with a word of caution: throughout this section, the proofs of the theorems and the analysis of the examples are long and technically involved. We state the results and provide proof sketches in the text below; all details are available in the WSM.

Identical Independent Tasks

First we consider identical independent tasks that can be executed concurrently. Recall that P is the total number of processors. We identify a task $\mathcal{T}$ with its type (i.e., set of parameters) $\mathcal{T}=(T_{base}, p, C, R)$: length T_{base}, number of processors p, checkpoint time C, recovery time R.

Theorem 22. *Consider n identical tasks of same type $\mathcal{T}=(T_{base},p,C,R)$ to be executed concurrently on $n \times p \leq P$ processors with individual failure rate $\lambda = \frac{1}{\mu_{ind}}$. The downtime is D. For the MinExp strategy, N_{ME} is the number of checkpoints, and W_{ME} is the length of each segment, as given by Definition 3. Let $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{ME}+C+\tilde{R})}$ be the probability of success of a segment with re-execution cost $\tilde{R}$ ($\tilde{R}=0$ if no re-execution, or $\tilde{R}=R$ otherwise), and $Q^* = \frac{1}{1-P_{suc}(R)}$. Let the ratio be $r_{id}^{ME}(n,\mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}}$, where $\mathbb{E}(T_{tot})$ is the expectation of the total time T_{tot} of the MinExp strategy. We have:*

$$r_{id}^{ME}(n,\mathcal{T}) \leq \left(\frac{\log_{Q^*}(n)}{N_{ME}} + \log_{Q^*}(\log_{Q^*}(n)) + 1 + \frac{\ln(Q^*)}{12N_{ME}} + \frac{1}{\ln(Q^*)N_{ME}}\right) \times \left(1 + \frac{C+R+D}{W_{ME}}\right) + \frac{C}{W_{ME}} + 1 + o(1). \tag{6.2}$$

Note that if n is small, the ratio holds by replacing all negative or undefined terms by 0.

Proof. First, a segment consists of the re-execution cost $\tilde{R}$, the work W_{ME} and the checkpoint cost C. Since failures may occur during recovery or checkpoint, the total processing time is $W_{ME} + \tilde{R} + C$. Thus, given the exponential failure probability, we have $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{ME}+C+\tilde{R})}$. The MinExp strategy is a $r_{id}^{ME}(n,\mathcal{T})$-approximation of the base time $T_{base} = N_{ME}W_{ME}$, hence also of the optimal expected execution time. Let M_f be the maximum number of failures over all tasks. We process N_{ME} segments of length $W_{ME}+C$, and each failure in a segment incurs an additional time upper bounded by $D+R+W_{ME}+C$. The expectation $\mathbb{E}(T_{tot})$ of the total time T_{tot} of the MinExp strategy is at most:

$$\mathbb{E}(T_{tot}) \leq T_{base} + N_{ME}C + \mathbb{E}(M_f)(W_{ME}+C+R+D),$$

hence

$$r_{id}^{ME}(n,\mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}} \leq 1 + \frac{C}{W_{ME}} + \frac{\mathbb{E}(M_f)}{N_{ME}}\left(1 + \frac{C+R+D}{W_{ME}}\right). \tag{6.3}$$

We continue with the computation of $\mathbb{E}(M_f)$. We first study the random variable (RV) N_f of the number of failures before completing a given task. We have identical segments $(s_1, s_2, ...)$ to process, each of them having a probability of success $p_{s_i} \in \{P_{suc}(R), P_{suc}(0)\}$, and we stop upon reaching the N_{ME} successes. Hence, s_1 is the first trial of the first segment; if s_1 succeeds, which happens with probability $P_{suc}(0)$, s_2 corresponds to the first trial of the second segment, and succeeds with probability $P_{suc}(0)$; otherwise, s_2 corresponds to the second trial of the first segment, and succeeds with probability $P_{suc}(R)$.

We are interested in the number of failures N_f before having N_{ME} successes. Clearly, if N'_f represents the RV for the same problem except that all segments have the same probability of success $P_{suc}(R)$, all segments are less likely or equally likely to succeed, and

$$\forall x, \mathbb{P}\{N'_f \leq x\} \leq \mathbb{P}\{N_f \leq x\}. \tag{6.4}$$

Now, let M'_f be the RV equal to the maximum of n IID (Independent and identically Distributed) RVs following N'_f. Equation (6.4) leads to $\mathbb{E}(M'_f) \geq \mathbb{E}(M_f)$. Each N'_f is a negative binomial RV with parameters $(N_{ME}, P_{suc}(R))$. We refine the analysis from [70] by bounding the sum of some Fourier coefficients. The details are technicals and not required for this thesis, but can be found in [R2, Appendix A] to show that:

$$\mathbb{E}(M'_f) \leq \log_{Q^*}(n) + (N_{ME}-1)\log_{Q^*}(\log_{Q^*}(n)) + N_{ME} + \left(\frac{\ln(Q^*)}{12} + \frac{1}{\ln(Q^*)}\right) + o(1). \tag{6.5}$$

Recall that $Q^* = \frac{1}{1-P_{suc}(R)}$. Here, we assume for convenience that $\log_{Q^*}(\log_{Q^*}(n)) \geq 0$, but otherwise we can replace it by 0 and the ratio holds. Plugging the bound of Equation (6.5) back into Equation (6.3) leads to Equation (6.2). □

We provide an informal simplification of the bound in Equation (6.2). Under reasonable settings, we have $C, D, R \ll \mu_{ind}$, and the probability of success P_{suc} of each segment is pretty high, hence $Q^* > e$. For this reason, we have (i) $\forall x, \log_{Q^*}(x) < \ln(x)$; (ii) $\frac{C+R+D}{W_{ME}} \approx 0$; (iii) $\frac{\ln(Q^*)}{12N_{ME}} \leq 1$; and (iv) $\frac{1}{\ln(Q^*)N_{ME}} \approx 0$. Altogether, the bound simplifies to:

$$r_{id}^{ME}(n, \mathcal{T}) \leq \frac{\ln(n)}{N_{ME}} + \ln(\ln(n)) + 3 + o(1). \tag{6.6}$$

Here is a more precise statement (again, the proof just uses simple maths and is omitted here, but it can be found in [R2, Appendix H]):

Proposition 2. *We have* $r_{id}^{ME}(n, \mathcal{T}) \leq \frac{4}{5}\left(\frac{\ln(n)}{N_{ME}} + \ln(\ln(n))\right) + 3 + \frac{3}{N_{ME}} + o(1)$ *under the following assumptions:*

- *A checkpoint of length C succeeds with probability at least 0.99;*
- $D \leq R \leq C$*;*
- *A segment of length W_{ME} fails with probability at least 10^{-10};*
- $T_{base} > 2(C + R + D)$ *(otherwise the tasks are so small that no checkpoints are needed).*

Tightness of the bound $r_{id}^{ME}(n, \mathcal{T})$ of Theorem 22

Consider a set $\mathcal{T}$ of n identical uni-processor tasks with $T_{base} = 2K - 1$, $C = 1$, $D = R = 0$ and $\lambda = \frac{\ln(1+\frac{1}{2K})}{2K}$ so that $e^{-\lambda(T_{base}+C)} = \frac{2K}{2K+1}$. Here, $K \geq 2$ is fixed, and n is the variable. We assume that all tasks execute in parallel, i.e., $P \geq n$. Under these settings, we show that $r_{id}^{ME}(n, \mathcal{T}) = \Theta(\ln(n))$, thereby showing the asymptotical tightness of the bound given in Theorem 22.

Arbitrary independent tasks

We now proceed with different independent tasks that can be executed concurrently:

Theorem 23. *Consider a set $\mathcal{T}$ of n tasks. The i-th task has profile $\mathcal{T}_i = (T^i_{base}, p_i, C_i, R_i)$. These tasks execute concurrently, hence $\sum_{i=1}^{n} p_i \leq P$. The individual fault rate on each processor is λ. The downtime is D. For the MinExp strategy, N^i_{ME} is the number of checkpoints, and W^i_{ME} is the length of each segment, for task i. Let $P^i_{suc}(R_i) = e^{-p_i\lambda(W^i_{ME}+C_i+R_i)}$ be the probability of success of a segment of task i with re-execution cost R_i, and $Q^*_i = \frac{1}{1-P^i_{suc}(R_i)}$. Then, the MinExp strategy is a $r^{ME}(n,\mathcal{T})$-approximation of the failure-free execution time, hence also of the optimal expected execution time, where:*

$$r^{ME}(n,\mathcal{T}) \leq 2 \max_{1\leq i\leq n} \left(r^{ME}_{id}(n,\mathcal{T}_i)\right). \tag{6.7}$$

The key element of this proof is a new result (to the best of our knowledge) on expectations of RVs:

Theorem 24. *Let $(X_1,\ldots,X_n)$ be n independent positive RVs with finite expectation, and let $Y = \max(X_1,\ldots,X_n)$. Let Z_i be the maximum of n IID RVs $X_{i,j}$ with the same law as X_i (thus, $Z_i = \max(X_{i,1}, X_{i,2}, \ldots, X_{i,n})$).*
Then, $\mathbb{E}(Y) \leq 2\max_i(\mathbb{E}(Z_i))$.

Proof. The key idea is to build a worse case scenario, and then show that any set of positive RVs may be transformed to this worse case scenario using "algorithmic steps" that may only increase $\frac{\mathbb{E}(Y)}{2\max_i(\mathbb{E}(Z_i))}$. The proof is however very long and is kept in Appendix A, in case the reviewers want to take a quick glance at it. □

In fact, Theorem 24 allows us to directly extend Theorem 22 to Theorem 23. For all i, let T_i be the Random Variable representing the execution time of the task with profile $\mathcal{T}_i$ and let $X_i = \frac{T_i}{T^i_{base}}$. Clearly, the X_i's are independent and positive, so $Y = \max(X_1,\ldots,X_n)$ matches the condition of Theorem 23. Furthermore, for any given i, we suppose that we were to schedule a shelf of n identical tasks of type $\mathcal{T}_i$ in which $T_{i,j}$ is the Random Variable representing their execution time and $X_{i,j} = \frac{T_{i,j}}{T^i_{base}}$. Then, clearly, the $X_{i,j}$'s are IID and follow the same law as X_i, thus we can define $Z_i = \max_j(X_{i,j})$ to reach the conditions of Theorem 24. Finally, the two following equations hold:

$$\begin{aligned} r^{ME}(n,\mathcal{T}) &= \mathbb{E}\left(\max_i\left(\frac{T_i}{\max_i(T^i_{base})}\right)\right) \\ &\leq \mathbb{E}\left(\max_i\left(\frac{T_i}{T^i_{base}}\right)\right) = \mathbb{E}(Y) \end{aligned}$$

$$\forall i, r^{ME}_{id}(n,\mathcal{T}_i) = \mathbb{E}\left(\max_j\left(\frac{T_{i,j}}{T^i_{base}}\right)\right) = \mathbb{E}\left(\max_j(X_{i,j})\right) = \mathbb{E}(Z_i)$$

We can then apply Theorem 24 and obtain the result:

$$r^{ME}(n,\mathcal{T}) \leq \mathbb{E}(Y) \leq 2\max_i(\mathbb{E}(Z_i)) = 2\max_{1\leq i\leq n}\left(r^{ME}_{id}(n,\mathcal{T}_i)\right). \tag{6.8}$$

Similarly to identical tasks, under reasonable assumptions, we derive a simplified bound:

$$r^{ME}(n,\mathcal{T}) \leq 2\frac{\ln(n)}{\min_{1\leq i\leq n}(N^i_{ME})} + 2\ln(\ln(n)) + 6 + o(1).$$

6.3.2 MinExp for Workflows

We proceed to the study of MinExp for a workflow of tasks, with task dependencies. We build upon the results for identical tasks (see Equation (6.2)), that can be reused for each task of the workflow.

Theorem 25. *Let $\mathbb{S}$ be a failure-free schedule of a workflow $\mathcal{W}$ of n tasks. The i-th task has profile $\mathcal{T}_i = (T^i_{base}, p_i, C_i, R_i)$. The individual fault rate on each processor is λ. The downtime is D. Let Δ be the maximum number of tasks processed concurrently by the failure-free schedule $\mathbb{S}$ at any instant. Then, the MinExp strategy is a $r^{ME}(\Delta, \mathcal{W})$-approximation of the failure-free execution time, where*

$$r^{ME}(\Delta, \mathcal{W}) \leq 2 \max_{1 \leq i \leq n} r^{ME}_{id}(\Delta, \mathcal{T}_i). \tag{6.9}$$

In other words, the degree of parallelism Δ of the schedule becomes the key parameter to bound the performance of the MinExp strategy, rather than the total number n of tasks in the workflow. Similarly to independent tasks, under reasonable assumptions, we derive a simplified bound:

$$r^{ME}(\Delta, \mathcal{W}) \leq 2\frac{\ln(\Delta)}{\min_{1 \leq i \leq n}(N^i_{ME})} + 2\ln(\ln(\Delta)) + 6 + o(1).$$

Proof. As stated in Section 6.2.2, we enforce the same ordering of starting times in the initial schedule $\mathbb{S}$ and in the failure-aware schedule $\mathbb{S}$' returned by MinExp: if task i starts after task j in $\mathbb{S}$, the same will hold in $\mathbb{S}$'. However, we greedily start a task as soon as enough processors are available, which may result in using different processors for a given task in $\mathbb{S}$ and $\mathbb{S}$'. Consider an arbitrary failure scenario, and let T_i be the execution time of task i in $\mathbb{S}$'. Let $T(\mathbb{S}')$ be the total execution time of $\mathbb{S}$'. We want to prove that:

$$\mathbb{E}(T(\mathbb{S}')) \leq 2 \max_{1 \leq i \leq n} r^{ME}_{id}(\Delta, \mathcal{T}_i) T(\mathbb{S}), \tag{6.10}$$

where $T(\mathbb{S})$ is the (deterministic) total execution time of $\mathbb{S}$.

To analyze $\mathbb{S}$', we partition $\mathbb{S}$ into a series of execution slices, where a slice is determined by two consecutive events. An event is either the starting time or the ending time of a task. Formally, let s_i be the starting time of task i in $\mathbb{S}$, and e_i be its ending time. We let $\{t_j\}_{0 \leq j \leq K} = \cup_{i=1}^{n} \{s_i, e_i\}$ denote the set of events, labeled such that $\forall j \in [0, K-1], t_j < t_{j+1}$. Note that we may have $K + 1 < 2n$ if two events coincide. We partition $\mathbb{S}$ into K slices S_j, $1 \leq j \leq K$, which are processed sequentially. Slice S_j spans the interval $[t_{j-1}, t_j]$. In other words, the length of S_j is $t_j - t_{j-1}$. Let $B_j \subset \mathcal{W}$ denote the subset of tasks that are (partially or totally) processed during slice S_j; note that $\Delta = \max_{j \in [1,K]} |B_j|$. Finally, for a task i in B_j, let $a_{i,j}$ be the fraction of the task that is processed during S_j (and let $a_{i,j} = 0$ if $i \notin B_j$).

As an example, we consider a workflow $\mathcal{W}$ consisting of $n = 4$ independent tasks, with $T^1_{base} = 6$, $T^2_{base} = 4$, $T^3_{base} = 8$ and $T^4_{base} = 9$. We have $P = 4$, $p_1 = p_2 = 2$ and $p_3 = p_4 = 1$. The optimal failure-free schedule $\mathbb{S}$ is shown in Figure 6.1, and has length 10. Note that task i is represented by its profile $\mathcal{T}_i$. There are five time-steps where an event occurs, thus $K = 4$ and $\{t_j\}_{0 \leq j \leq K} = \{0, 6, 8, 9, 10\}$. Therefore, $\mathbb{S}$ is decomposed into four slices, S_1 running in $[0, 6]$, S_2 in $[6, 8]$, S_3 in $[8, 9]$, and S_4 in $[9, 10]$. The $(a_{i,j})_{i \in [1,n], j \in [1,K]}$ are represented in brackets. Finally, $B_1 = \{1, 3, 4\}$, $B_2 = \{2, 3, 4\}$, $B_3 = \{2, 4\}$, $B_4 = \{4\}$, and $\Delta = 3$. We use the decomposition into slices to define a virtual schedule $\mathbb{S}^{virt}$, which consists of scaling the slices S_j to account for failures in $\mathbb{S}$'. For each slice S_j, the scaling is the largest ratio $\frac{T_i}{T^i_{base}}$ over all tasks $i \in B_j$. Hence, $\mathbb{S}^{virt}$ is composed of K slices S^{virt}_j

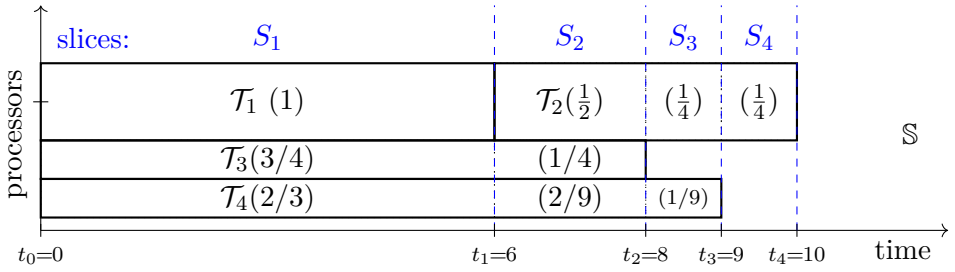


Figure 6.1: Example for the proof of Theorem 25: Schedule $\mathbb{S}$.

whose length is $T(S_j^{virt}) = \left(\max_{i\in B_j} \frac{T_i}{T_{base}^i}\right) T(S_j)$. Within each slice S_j^{virt}, for each task $i \in B_j$, we execute the same fraction $a_{i,j}$ of task i as in the original schedule $\mathbb{S}$, for a duration $a_{i,j}T_i$, so that some tasks in B_j may not execute during the whole length of S_j^{virt}, contrarily to during the initial schedule $\mathbb{S}$. The $(a_{i,j})_{i\in[1,n],j\in[1,K]}$ for the example in the proof of Theorem 25 are given by

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & 1/4 & 1/4 \\ 3/4 & 1/4 & 0 & 0 \\ 2/3 & 2/9 & 1/9 & 0 \end{bmatrix}$$

The total execution time of $\mathbb{S}^{virt}$ is $\sum_{j=1}^{K} T(S_j^{virt})$. From Theorem 23, we directly have that

$$\begin{aligned} \mathbb{E}(T(S_j^{virt})) &\leq 2\max_{i\in B_j}(r_{id}^{ME}(|B_j|, \mathcal{T}_i))T(S_j) \\ &\leq 2\max_{1\leq i\leq n}(r_{id}^{ME}(\Delta, \mathcal{T}_i))T(S_j) \end{aligned}$$

The second inequality holds because $|B_j| \leq \Delta$ and because r_{id}^{ME} increases when the number of tasks increases. Finally,

$$\begin{aligned} \mathbb{E}(T(\mathbb{S}^{virt})) &= \sum_{j=1}^{K} \mathbb{E}(T(S_j^{virt})) \\ &\leq 2\max_{1\leq i\leq n}(r_{id}^{ME}(\Delta, \mathcal{T}_i)) \sum_{j=1}^{K} T(S_j), \end{aligned}$$

where $\sum_{j=1}^{K} T(S_j) = T(\mathbb{S})$.

Finally, here is the proof by induction to prove that no task starts nor ends later in $\mathbb{S}$' than in $\mathbb{S}^{virt}$. Recall that the key element is that the ordering of starting times from $\mathbb{S}$ is preserved in both $\mathbb{S}^{virt}$ and $\mathbb{S}'$.

- The first task starts at time 0 for both $\mathbb{S}^{virt}$ and $\mathbb{S}'$, and may be suspended in $\mathbb{S}^{virt}$, hence its ending time cannot be larger in $\mathbb{S}'$;
- Let $i > 1$, and suppose the induction hypothesis holds for the first $i-1$ tasks. Let x be the starting time of task i in schedule $\mathbb{S}^{virt}$. By definition of the slices in $\mathbb{S}^{virt}$, when a task starts, all the unfinished tasks are running concurrently; thus there are enough processors to process task i and all the unfinished tasks among the first $i-1$ ones. Since the ending time of all these unfinished tasks may not be larger in $\mathbb{S}'$, there are enough processors to start task i in $\mathbb{S}$' at time x if it has not started yet. Because we try to start the i-th task in $\mathbb{S}$' before the ones that are started later in $\mathbb{S}$, we will indeed not start task i later in $\mathbb{S}'$ than in $\mathbb{S}^{virt}$. And because task i will not be suspended in $\mathbb{S}'$, it will not end later either.

This concludes the induction and the proof of Theorem 25.

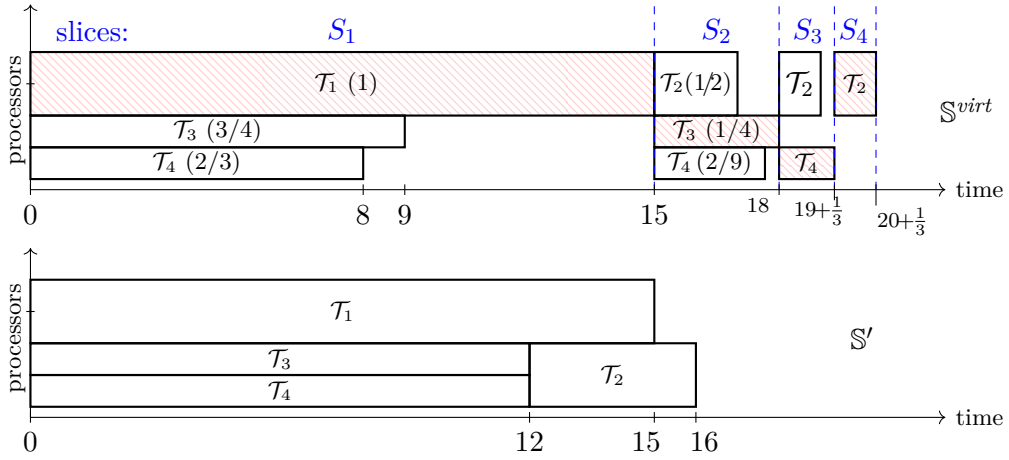


Figure 6.2: Schedules $\mathbb{S}^{virt}$ (top) and $\mathbb{S}'$ (bot.) for the example.

Going back to the example, assume that $T_1 = 15$, $T_2 = 4$, and $T_3 = T_4 = 12$ in $\mathbb{S}$'. Then, we obtain the task with the largest ratio $\frac{T_i}{T_{base}^i}$ for each slice: task 1 for S_1, task 3 for S_2, task 4 for S_3, and task 2 for S_4. The schedule $\mathbb{S}^{virt}$ is shown at the top of Figure 6.2 and has length $T(\mathbb{S}^{virt}) = 20 + 1/3$ (and the tasks with largest ratio in each slice are hatched). Finally, the schedule $\mathbb{S}'$ is shown at the bottom of Figure 6.2, and $T(\mathbb{S}') = 16$. □

We point out that Theorem 25 applies to workflows with arbitrary dependences, and with rigid or moldable tasks. The bound given for $r^{ME}(\Delta, \mathcal{W})$ is relative to the execution time of the failure-free schedule. If this failure-free schedule is itself a ρ-approximation of the optimal solution, then we have derived a $r^{ME}(\Delta, \mathcal{W}) \times \rho$ approximation of the optimal solution.

6.4 The CheckMore Strategies

The previous section has shown that, in the presence of failures, the ratio of the actual execution time of a workflow over its failure-free execution time, critically depends upon the maximum degree of parallelism Δ achieved by the initial schedule.

In this section, we introduce CheckMore strategies, which checkpoint workflow tasks more often than MinExp, with the objective to decrease the ratio above. The number of checkpoints for each task becomes a function of the degree of parallelism in the execution. We define SafeCheck(δ), the number of checkpoints for a task, given a parameter δ (typically the degree of parallelism):

Definition 4. *SafeCheck(δ) partitions a parallel task of length T_{base}, with p processors and checkpoint time C, into $N_{SC}(\delta) = \left\lceil \frac{(\ln(\delta)+1)T_{base}}{W_{YD}} \right\rceil$ equal-length segments, each followed by a checkpoint, where $W_{YD} = \sqrt{\frac{2C}{p\lambda}} = \sqrt{\frac{2\mu_{ind}C}{p}}$. Each segment is of length $W_{SC}(\delta) = \frac{T_{base}}{N_{SC}(\delta)}$.*

Because $N_{SC}(1) = N_{ME}$, MinExp corresponds to applying SafeCheck (1) to all tasks. The key building block of the analysis of MinExp is Theorem 22 for identical independent tasks. The good news is that Theorem 22 holds for any checkpointing strategy, not just for the Young/Daly approach, and can easily be extended if each task is checkpointed following SafeCheck(δ):

Theorem 26. *Consider n identical tasks of same type $\mathcal{T}$=(T_{base},p,C,R) to be executed concurrently on $n \times p \leq P$ processors with individual failure rate $\lambda = \frac{1}{\mu_{ind}}$. The downtime is D. For the* SafeCheck(δ) *strategy, $N_{SC}(\delta)$ is the number of checkpoints, and $W_{SC}(\delta)$ is the length of each segment, as given by Definition 3. Let $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{SC}+C+\tilde{R})}$ be the probability of success of a segment with re-execution cost $\tilde{R}$ ($\tilde{R} = 0$ if no re-execution, or $\tilde{R} = R$ otherwise), and $Q^* = \frac{1}{1-P_{suc}(R)}$. Let $r_{id}^{SC}(\delta, n, \mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}}$, where $\mathbb{E}(T_{tot})$ is the expectation of the total time T_{tot} of the* SafeCheck(δ) *strategy. Then:*

$$r_{id}^{SC}(\delta, n, \mathcal{T}) \leq \left(\frac{\log_{Q^*}(n)}{N_{SC}(\delta)} + \log_{Q^*}(\log_{Q^*}(n)) + 1 + \frac{\ln(Q^*)}{12N_{SC}(\delta)} + \frac{1}{\ln(Q^*)N_{SC}(\delta)}\right) \times \left(1 + \frac{C+R+D}{W_{SC}(\delta)}\right) + \frac{C}{W_{SC}(\delta)} + 1 + o(1). \tag{6.11}$$

Note that if n is small, the ratio holds by replacing all negative or undefined terms by 0.

To prove this theorem, we reuse the proof of Theorem 22: we just need to replace N_{ME} by $N_{SC}(\delta)$, and W_{ME} by $W_{SC}(\delta)$.

The idea behind SafeCheck(δ) is the following: when processing δ jobs in parallel, the expected maximum number of failures given by Equation (6.5) eventually grows proportionally to its first term, $\log_{Q^*}(\delta)$, which is $\Theta(\ln(\delta))$. To accommodate this growth, we reduce the segment length by a factor $\ln(\delta)$, so that the total failure-induced overhead does not increase much. This is exactly what SafeCheck(δ) does, when δ tasks are processed in parallel. Similarly, the first term $\frac{\log_{Q^*}(n)}{N_{ME}(n)}$ of the ratio in Equation (6.2) was dominant for MinExp, while it becomes almost constant in Equation (6.11). To that extent, CheckMore generalizes this idea to general workflows using SafeCheck(δ) as a subroutine. We provide two variants of CheckMore:

Definition 5. *Consider a failure-free schedule $\mathbb{S}$ for a workflow $\mathcal{W}$ of n tasks:*

- *The* CheckMore *algorithm applies* SafeCheck(Δ_i) *to each task i, where Δ_i is the largest number of tasks that are executed concurrently during the processing of task i.*
- *The* BasicCheckMore *algorithm applies* SafeCheck($\min(n, P)$) *to all tasks, where P is the number of processors.*

The main reason for introducing BasicCheckMore is that we do not need to know the maximum degree Δ of parallelism in $\mathbb{S}$ to execute BasicCheckMore (because we always have $\Delta \leq \min(n, P)$. In fact, we do not even need to know the failure-free schedule for BasicCheckMore (contrarily to CheckMore), we just need an ordered list of tasks and to greedily start them in this order.

Theorem 27. *Let $\mathbb{S}$ be a failure-free schedule of a workflow $\mathcal{W}$ of n tasks. The i-th task has profile $\mathcal{T}_i = (T_{base}^i, p_i, C_i, R_i)$. Let Δ_i be the maximum number of tasks processed concurrently to task i by $\mathbb{S}$ at any instant, and let $\Delta = \max_{1 \leq i \leq n} \Delta_i$. Then* CheckMore *is a $r^{CM}((\Delta_i)_{i \leq n}, \mathcal{W})$-approximation of the failure-free execution time in expectation:*

$$r^{CM}((\Delta_i)_{i \leq n}, \mathcal{W}) \leq 2 \max_{1 \leq i \leq n} r_{id}^{SC}(\Delta_i, \Delta_i, \mathcal{T}_i). \tag{6.12}$$

And BasicCheckMore *is a $r^{BCM}(\min(n, P), \mathcal{W})$-approximation of the failure-free execution time in expectation:*

$$r^{BCM}(\min(n, P), \mathcal{W}) \leq 2 \max_{1 \leq i \leq n} r_{id}^{SC}(\min(n, P), \Delta, \mathcal{T}_i). \tag{6.13}$$

Proof. To prove this theorem, we just need to adapt the proof of Theorem 25. In fact, the analysis in Theorem 25 did not depend upon the checkpoint strategy, thus using

the same slices $(S_j)_{j\in[1,K]}$ and virtual schedule $\mathbb{S}^{virt}$, we have for both CheckMore and BasicCheckMore:

$$\mathbb{E}(T(\mathbb{S}')) \leq \mathbb{E}(T(\mathbb{S}^{virt})) = \sum_{j=1}^{K} \mathbb{E}(T(S_j^{virt})). \tag{6.14}$$

Again, for all slices S_j and all tasks $i \in B_j$ (recall that B_j is the set of tasks in slice S_j), we have $X_i = \frac{T_i}{T_{base}^i}$. Then, the scaling of S_j, $\frac{T(S_j^{virt})}{T(S_j)}$, corresponds to $\max_{i\in B_j} X_i$. We then can safely use Theorem 24; assuming that each task i is checkpointed according to SafeCheck(δ_i), we obtain:

$$\begin{aligned} \tfrac{\mathbb{E}(T(S_j^{virt}))}{T(S_j)} &\leq 2\max_{i\in B_j}\left(r_{id}^{SC}(\delta_i, |B_j|, \mathcal{T}_i)\right) \\ &\leq 2\max_{1\leq i\leq n}\left(r_{id}^{SC}(\delta_i, \Delta_i, \mathcal{T}_i)\right) \end{aligned} \tag{6.15}$$

Finally, using Equation (6.15), Equation (6.14), and $\forall i, \Delta_i \leq \Delta$, we obtain for CheckMore and BasicCheckMore respectively:

$$\mathbb{E}(T(\mathbb{S}')) \leq 2\max_{1\leq i\leq n}\left(r_{id}^{SC}(\Delta_i, \Delta_i, \mathcal{T}_i)\right)\sum_{j=1}^{K} T(S_j);$$

$$\mathbb{E}(T(\mathbb{S}')) \leq 2\max_{1\leq i\leq n}\left(r_{id}^{SC}(min(n,P), \Delta, \mathcal{T}_i)\right)\sum_{j=1}^{K} T(S_j). \qquad \square$$

Note that for all i, $\Delta_i \leq \Delta$ and $\Delta_i \leq min(n, P)$, so it is extremely likely that the bound obtained for r^{CM} is smaller than the one obtained for r^{BCM}. To illustrate the difference between the bounds of CheckMore and MinExp, we have also shown in this work that for a shelf of n identical uni-processor tasks running in parallel, r_{id}^{CM} is an order of magnitude lower than r_{id}^{ME} under reasonable assumptions and when n is large enough. Again, the details are not needed for this thesis (see [R2, Appendix H]).

We conclude this section by returning to the example of Section 6.3.1, and showing that CheckMore (equivalent to BasicCheckMore in this case) can be arbitrarily better than MinExp, for reasonable tasks when their number is large enough. This is not surprising because the larger n is, the more important it is to checkpoint more. Therefore, we omit the proof in this thesis (but it can be found in [R2, Appendix G.2]).

Proposition 3. *Consider a set $\mathcal{T}$ of $n(K)$ identical uni-processor tasks with type $\mathcal{T} = (2K-1, 1, 10)$, $D = 0$ and $\lambda(K) = \frac{\ln(1+\frac{1}{2K})}{2K}$. We assume that all tasks execute in parallel, i.e., $P \geq n(K)$. When letting $n(K) = \lfloor e^{\sqrt{2/\lambda(K)-1}} \rfloor$ (hence $\ln(n(K)) = \Theta(K)$), and K tending to infinity, we have $r^{ME}(n(K), \mathcal{T}) = \Theta\left(\frac{K}{\ln(K)}\right)$ and $r^{BCM}(n(K), \mathcal{T}) = \Theta(1)$.*

6.5 Experimental Evaluation

We evaluate the performance of the different checkpointing strategies through simulations. We describe the simulation setup in Section 6.5.1, present the main performance comparison results in Section 6.5.2, and assess the impact of different parameters on the performance in Section 6.5.3. We further provide some performance statistics in Section 6.5.4 and conclude with a brief summary in Section 6.5.5. Our in-house simulator is written in C++ and is publicly available for reproducibility purpose in `https://graal.ens-lyon.fr/~yrobert/simulator.zip`.

6.5.1 Simulation Setup

We evaluate and compare the performance of the three checkpointing strategies MinExp, CheckMore and BasicCheckMore. All strategies are coupled with a failure-free schedule computed by a list scheduling algorithm (see below). The workflows used for evaluation are generated from WorkflowHub [62] (formerly Pegasus [154]), which offers realistic synthetic workflow traces with a variety of characteristics and they have been shown to accurately resemble the ones from real-world workflow executions [6, 62]. Specifically, we generate the following nine different types of workflows offered by WorkflowHub that model applications in various scientific domains:

- Blast: a bioinformatics workflow for searching biological sequence databases and identifying amino-acid or DNA sequences that resemble query sequences;
- Bwa: a bioinformatics workflow for performing DNA sequence alignment using the "Burrows-Wheeler Aligner";
- Cycles: an agroecosystem workflow for conducting simulations of crop production and water, carbon and nitrogen cycles in the soil-plant-atmosphere continuum;
- Epigenomics: a bioinformatics workflow for automating various operations in genome sequence processing;
- Genome: a bioinformatics workflow for identifying mutational overlaps to provide statistical evaluation of potential disease-related mutations;
- Montage: an astronomy workflow for analyzing multiple input images to create custom mosaics of the sky;
- Seismology: a seismology workflow for performing seismogram deconvolutions to estimate earthquake source time functions;
- SoyKB: a bioinformatics workflow for performing large-scale next-generation sequencing of soybean lines within the Soybean Knowledge Base (SoyKB);
- Sras: a bioinformatics workflow for downloading and aligning data in the Sequence Read Archive (SRA).

Each trace defines the general structure of the workflow, whose number of tasks and total execution time can be specified by the user[2]. All tasks generated in WorkflowHub are uni-processor tasks.

In the experiments, we evaluate the checkpointing strategies under the following parameter settings:

- Number of processors: $P = 2^{14} = 16384$;
- Checkpoint/recovery/down time: $C = R = 1$ min, $D = 0$;
- MTBF of individual processor: $\mu_{ind} = 10$ years;
- Number of tasks of each workflow: $n \approx 50000$.

Furthermore, the total failure-free execution times of all workflows are generated such that they complete in 3-5 days. This is typical of the large scientific workflows that often take days to complete as observed in some production log traces [5, 127]. To demonstrate the robustness of our evaluation, we also generate small workflow traces that take less than a day (i.e., 15 - 24 hours) to complete. This is roughly one fifth of the size of large workflows. As a result, to keep the average number of failures per task the same, we also scale the individual MTBF from 10 years to 2 years, while all the other parameters are kept the same. Section 6.5.2 will present the comparison results of different checkpointing strategies under the above parameter settings. In Section 6.5.3, we will further evaluate the impacts of different parameters (i.e., P, C, μ_{ind} and n) on the performance.

[2]Note that the workflow generator may offer a different number of tasks so as to guarantee the structure of the workflow. The difference, however, is usually small.

The evaluation methodology is as follows: for each set of parameters and each type of workflow trace, we generate 30 different workflow instances and compute their failure-free schedules. We use the list scheduling algorithm that orders the tasks using the Longest Processing Time (LPT) first policy: if several tasks are ready and there is at least one processor available, the longest ready task is assigned to the available processor to execute. Since all tasks are uni-processor tasks, LPT is known to be a 2-approximation algorithm [71]; also, LPT is known to be a good heuristic for ordering the tasks [108]. This order of execution will be enforced by all the checkpointing strategies. For each workflow instance, we further generate 50 different failure scenarios. Here, a failure scenario consists of injecting random failures to the tasks by following the Exponential distribution as described in Section 6.2.1. The same failure scenario will then be applied to each checkpointing strategy to evaluate its execution time for the workflow. We finally compute the *ratio* of a checkpointing strategy under a particular failure scenario as $\frac{T}{T_{base}}$, where T_{base} is the failure-free execution time of the workflow, and T is the execution time under the failure scenario. The statistics of these $30 \times 50 = 1500$ experiments are then compared using boxplots (that show the mean, median, and various percentiles of the ratio) for each checkpointing strategy. The boxes bound the first to the third quantiles (i.e., 25th and 75th percentiles), the whiskers show the 10th percentile to the 90th percentile, the black lines show the median, and the stars show the mean.

6.5.2 Performance Comparison Results

Figure 6.3 (left) shows the boxplots of the three checkpointing strategies in terms of their ratios for the nine different workflows, when their failure-free execution time is 3-5 days (i.e., large workflows).

First, we observe that CheckMore and BasicCheckMore have very similar performance, which in most cases are indistinguishable. This shows that BasicCheckMore offers a simple yet effective solution without the need to inspect the failure-free schedule, thus making it an attractive checkpointing strategy in practice. Also, both versions of CheckMore perform significantly better and with less variation than MinExp, except for the few workflows where the ratios of all strategies are very close to 1 (e.g., Bwa, SoyKB, Sras). Overall, the 90th percentile ratio of CheckMore never exceeds 1.08, whereas that of MinExp is much higher for most workflows and reaches almost 1.5 for Montage. Similarly, the average ratio of CheckMore never exceeds 1.03, while that of MinExp is again significantly higher and reaches more than 1.2 for Seismology and Montage.

We now examine a few workflows more closely to better understand the performance. For Sras, MinExp is slightly better than CheckMore, but the ratios of all strategies are near optimal (i.e., <1.003). In this workflow, very few tasks are extremely long while many others are very short, and there are very few dependencies among them. Thus, failures hardly ever hit the long tasks due to their few number, while failures that hit short tasks have little impact on the overall execution time. This is why the ratio is so small for all strategies. It also explains why MinExp outperforms CheckMore: although the maximum degree of parallelism is important, only a few tasks matter and they should be checkpointed à la Young/Daly to minimize their own expected execution time, and thereby that of the entire workflow. SoyKB and Bwa also have very low ratios. In the case of SoyKB, there is just not enough parallelism during the majority of the execution time, so all strategies are making reasonable checkpointing decisions, with CheckMore performing slightly better for taking into account this small parallelism. Bwa, on the other hand, has two source tasks that must be executed first and two sink tasks that must be executed last. Among them, one source task and one sink task are extremely long, so failures in

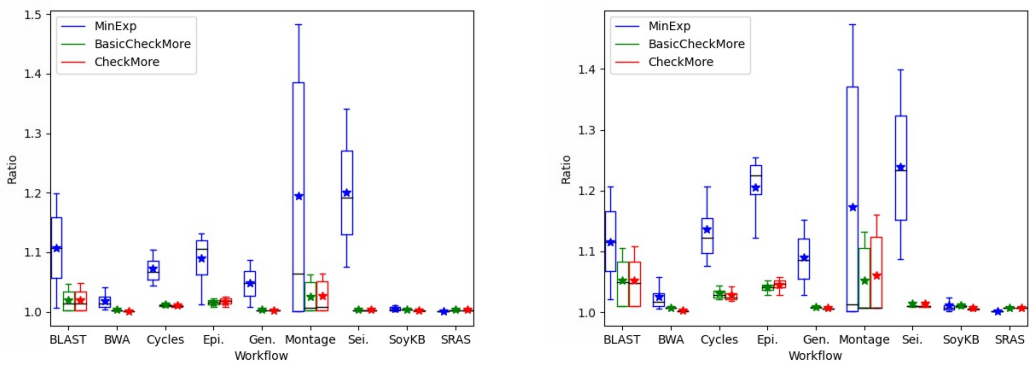


Figure 6.3: Performance (ratio) comparison of the three checkpointing strategies for the nine different workflows, with large workflows (i.e., failure-free execution time of 3-5 days) shown on the left, and small workflows (i.e., failure-free execution time of 15-24 hours) on the right.

other tasks have little impact (as in the case of SRAS). Yet the small tasks are not totally negligible here, because the dominant sink task must be processed after all of them, so it is still worth to optimize these tasks with CHECKMORE, which explains why it is slightly better than MINEXP.

For all the other workflows, CHECKMORE performs better than MINEXP by a significant margin. This is due to CHECKMORE's more effective checkpointing strategies given the specific structure of these workflows. For instance, MONTAGE has some key tasks that are dominant, so a failure that strikes most of the other tasks does not impact the overall execution time. This is similar to the case of SRAS and explains why, for all strategies, the first quantile of the ratio is very low (i.e., around 1). However, when a failure does strike one of the key tasks, the execution time will be heavily impacted. The difference with SRAS is that MONTAGE contains more key tasks that can run in parallel, so it is much more likely that one of them will fail, which is why checkpointing them with CHECKMORE is better. Next, BLAST and SEISMOLOGY have some source and sink tasks (as BWA), which, however, are not so dominant in length, making the difference between CHECKMORE and MINEXP higher even from the first quantile. Other workflows also have similar structures, which eventually contribute to the better performance of CHECKMORE over MINEXP.

Figure 6.3 (right) further shows the comparison results for the nine workflows when their failure-free execution time is 15-24 hours (i.e., small workflows). We can observe that the results are very similar to those for large workflows, which demonstrates that the relative performance of the three checkpointing strategies is not affected by the size of the workflows, provided that the average number of failures per task remains the same. Thus, in the subsequent experiments, we will only report results for the large workflows.

6.5.3 Impact of Different Parameters

We now study the impact of different parameters on the performance of the checkpointing strategies. In each set of experiments below, we vary a single parameter while keeping the others fixed at their base values. The results are shown in Figures 6.4-6.7, where the scale of the y-axis is kept the same for ease of comparison. For some figures with really small values, zoomed-in plots are also provided on the original figure for better viewing.

Impact of Number of Processors (P). We first assess the impact of the number of

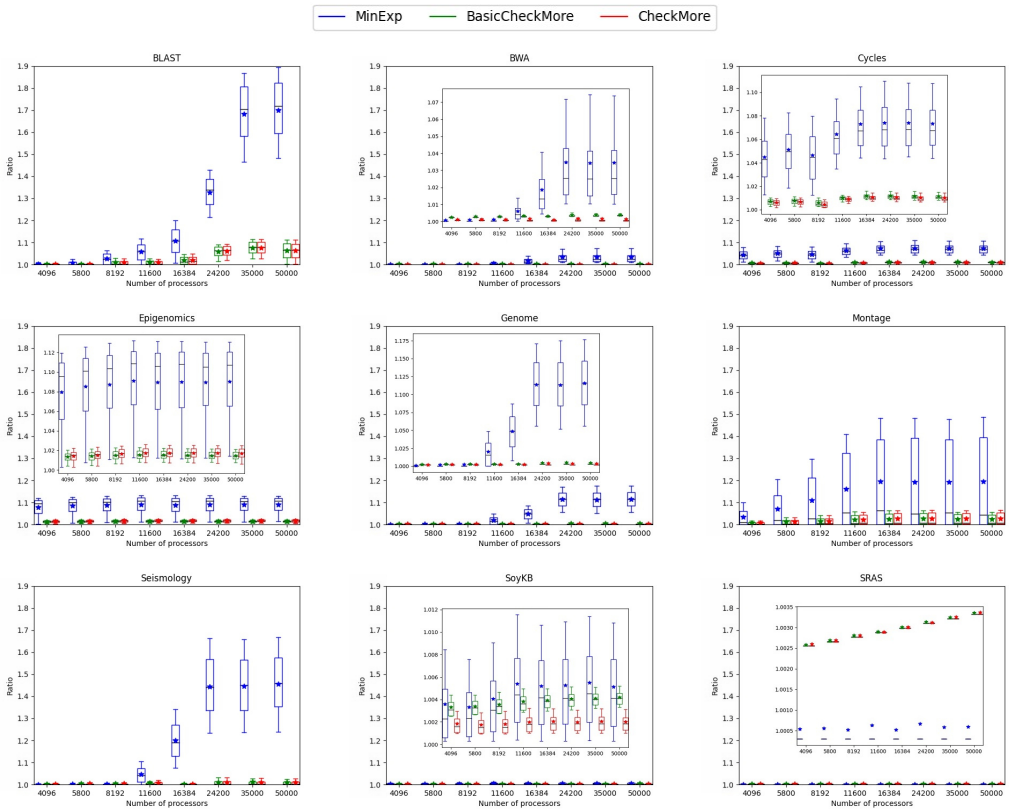


Figure 6.4: Impact of the number of processors (P) on the performance of the checkpointing strategies for different workflows.

processors, which is varied between 4096 and 50000, and the results are shown in Figure 6.4. In general, increasing the number of processors increases the ratio. This corroborates our theoretical analysis, because for most types of workflows, having more processors means having a larger Δ and thus a larger potential ratio, until P surpasses the width of the dependence graph. However, CkptMore and BasicCkptMore appear less impacted than MinExp.

For Blast, Bwa, Genome, Seismology, the ratio is very low when P is small for all checkpointing strategies. In fact, for these workflows, most tasks are quite independent. Thus, when n is large compared to P, even if a failure strikes a task, it will have little impact on the starting times of the other tasks. This is because we only maintain the order of execution but do not stick to the same mapping as in the failure-free schedule. For this reason, it is better to minimize each task's own execution time by using MinExp (i.e., CkptMore checkpoints a bit too much). However, when P becomes large, the performance of MinExp degrades significantly, with an average ratio even reaching 1.7 for Blast at $P = 50000$, whereas it stays below 1.1 for CkptMore.

For Epigenomics, Cycles and Montage, the ratio does not vary significantly with the number of processors, but is not negligible for MinExp even when P is small (between 1.05 and 1.2 depending on the workflow). For these workflows, the ratio of MinExp is 4 to 10 times higher than that of CkptMore, demonstrating the advantage of the latter checkpointing strategy.

Finally, for Sras, as the number of dominating tasks that could be run in parallel is way less than 4096, the ratio of MinExp does not vary much with P, while that of CkptMore increases with P as it tends to checkpoint more with an increasing number

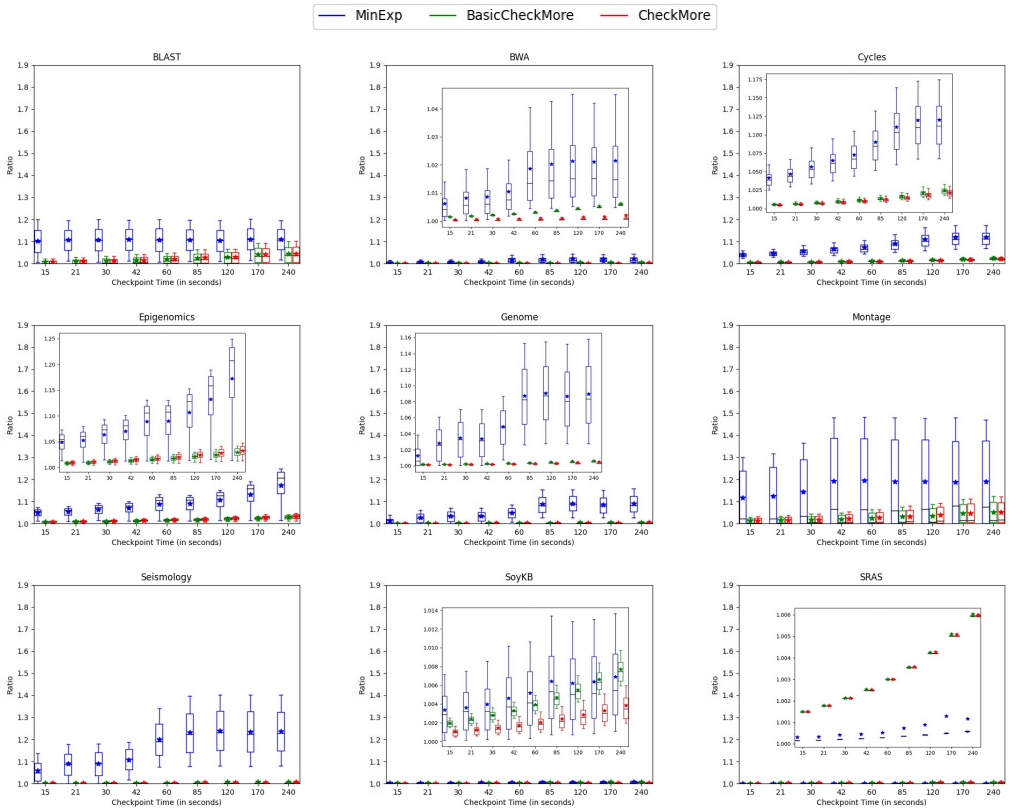


Figure 6.5: Impact of the checkpoint time (C) on the performance of the checkpointing strategies for different workflows.

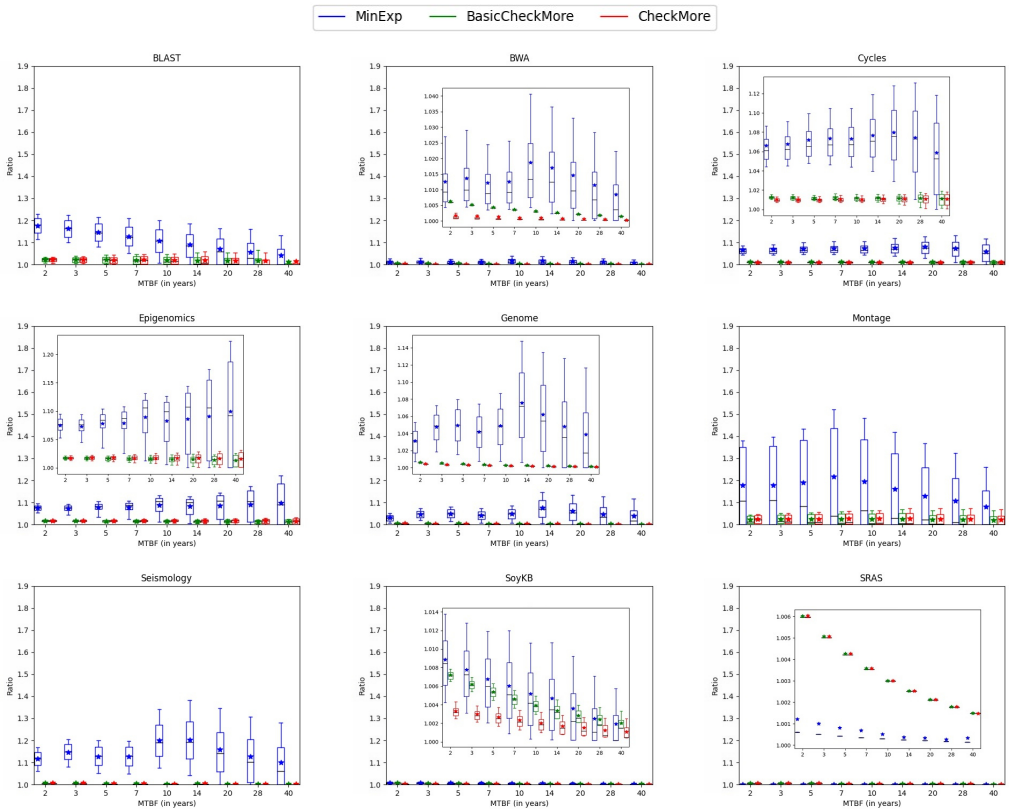


Figure 6.6: Impact of the individual MTBF (μ_{ind}) on the performance of the checkpointing strategies for different workflows.

of processors. Also, in more than 90% of the cases, the failures have strictly no impact on the overall execution time, since they do not hit the dominating tasks. This is why the average ratio is above the 90th percentile for all checkpointing strategies. Similarly, for SoyKB, the ratio is not impacted much for MinExp and CheckMore, especially for $P \geq 11000$.

Impact of Checkpoint Time (C). We now evaluate the impact of the checkpoint time by varying it between 15 and 240 seconds, and the results are shown in Figure 6.5. The ratio generally increases with C; this is consistent with Equation (6.2). when $R = C$ and $D = 0$, the approximation ratio satisfies $r \leq \left(\frac{X}{N} + Y\right)\left(\frac{2C}{W} + 1\right) + \frac{C}{W} + Z$, where X, Y and Z barely depend on C, N decreases with C, and $\frac{C}{W} \approx \sqrt{\frac{C}{2\mu}}$ increases with C. Intuitively, the checkpoint time impacts the ratio in two ways. First, as C increases, we pay more for each checkpoint, which could lead to an increased ratio. Second, as we use $W_{YD} = \sqrt{\frac{2C}{p\lambda}}$ to determine the checkpointing period and hence the number of checkpoints, a task will become less safe when C increases, because it will be checkpointed less, and this could also increase the ratio.

For example, looking at Genome under MinExp, we can see a clear increase in the ratio when C increases from 15 to 21. This is because the typical number of checkpoints for the critical tasks (that affect the overall execution time the most) drops from 3 to 2, thus the time wasted due to a failure increases from 33% to 50%. As C increases from 60 to 85, the typical number of checkpoints of these tasks further drops from 2 to 1, making the waste per failure increase to 100%, and so the ratio also greatly increases. For values of C between 21 and 42, even if the number of checkpoints does not change, the ratio increases

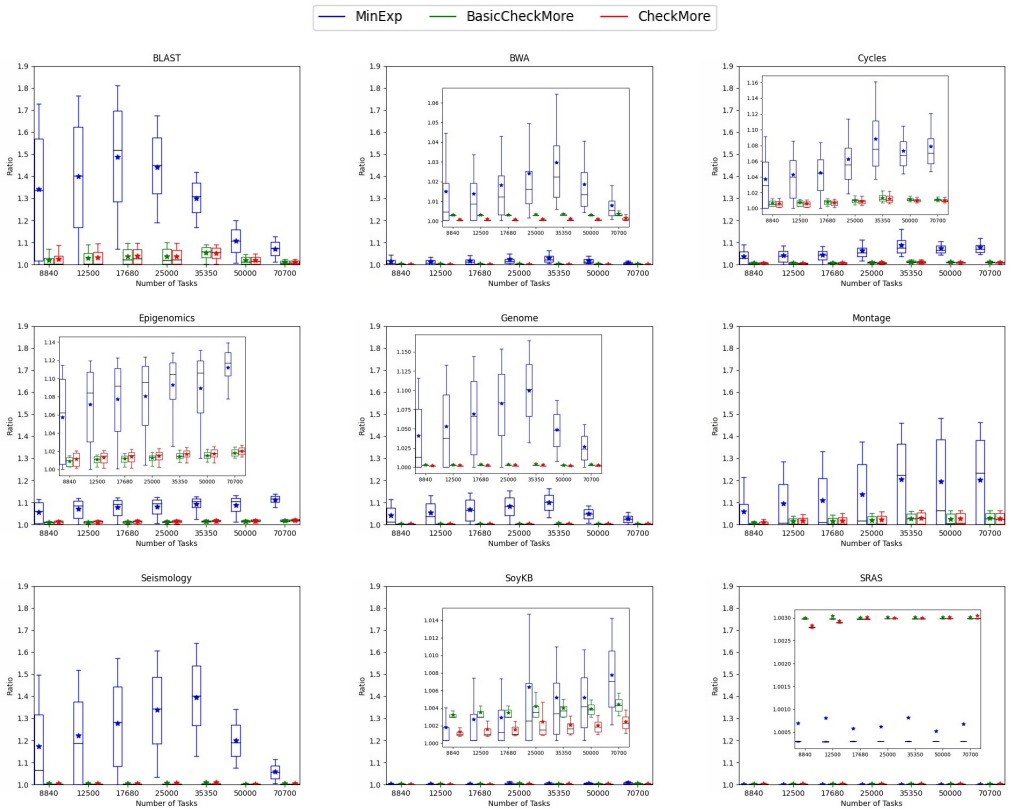


Figure 6.7: Impact of the number of tasks (n) on the performance of the checkpointing strategies for different workflows.

smoothly due to the increase in checkpoint time. The ratio of CheckMore, on the other hand, only increases slightly with the checkpoint time, which is, however, not visible in the figure due to the small values. Some other workflows, such as Bwa, Montage and Seismology, also clearly illustrate these phenomena.

For the remaining workflows, we can again see the impact of these two factors or their combination on the ratio. For instance, as most failures in Sras does not affect the overall execution time, the ratio of all strategies is only impacted by the checkpoint time. For Blast under MinExp, because most tasks are short and we have a single checkpoint to start with, the increase in checkpoint time is negligible compared to the waste induced by failures.

Impact of Individual MTBF (μ_{ind}). We evaluate the impact of individual processor's MTBF by varying it between 30 months and 40 years, and the results are shown in Figure 6.6. Intuitively, when μ_{ind} increases (or equivalently, the failure rate λ decreases), we would have fewer failures and expect the ratio to decrease. This is generally true for CheckMore but not always for MinExp. To understand why, we refer again to the simplified approximation ratio $r \leq \left(\frac{X}{N} + Y\right)\left(\frac{2C}{W} + 1\right) + \frac{C}{W} + Z$, where X, Y and Z are barely affected by μ_{ind}. Here, when the number of failures decreases, $W_{YD} = \sqrt{\frac{2C}{p\lambda}}$ increases, so the number of checkpoints decreases and the time wasted for each failure increases. This could potentially lead to an increase in the ratio. To illustrate this compound effect, we again look at Genome under MinExp. When μ_{ind} goes from 2.5 to 3.5 years, the typical number of checkpoints for the critical tasks (that affect the overall execution time the most) drops from 3 to 2, which increases the waste per failure by around 50%. This

together with the fact that MinExp does not take into account the parallelism results in an increase in the ratio. When μ_{ind} goes from 3.5 to 7 years, the ratio decreases simply because we have fewer failures. As μ_{ind} continues to increase to 14 years, the number of checkpoints for the critical tasks further drops from 2 to 1. This increases the waste per failure to 100%, which again leads to an increase in the ratio. From this point on, the ratio will just decrease with μ_{ind}, again due to fewer failures. The same phenomenon can be observed for some other workflows, such as Bwa, Montage and Seismology.

In yet some other workflows, the ratio simply decreases with μ_{ind}, such as for Blast and Sras. For Blast, even when μ_{ind} is small, we only checkpoint once, so the ratio decreases due to fewer failures. For Sras, failures usually do not impact the overall execution time, so the decrease in ratio is mainly due to the decrease in the number of checkpoints.

Finally, it is worth noting that the ratio variance increases as μ_{ind} increases. This is because when there are only a few failures and the length of the segments is large, the failure location (inside the segments) will matter significantly, especially for MinExp.

Impact of Number of Tasks (n). Finally, we study the impact of the number of tasks in the workflow, which is varied between 8800 and 70000, and the results are shown in Figure 6.7. Again, the ratio is impacted by the number of tasks in two different ways. First, when n increases, the width of the graph increases and so does Δ, and this would increase the ratio according to our analysis. Second, when n increases and P is fixed, the average number of tasks executed by each processor increases. This means that, if a failure occurs early in the execution, it is less likely to have a significant impact on the ratio, since multiple other tasks will be processed afterwards to balance the load, especially if the tasks are relatively independent.

These two phenomena are clearly observed in Blast under MinExp. This workflow mainly consists of a large batch of independent tasks. When n increases to 17680, which is approximately the number of processors ($P = 16384$), the ratio increases because Δ increases. After that, the ratio starts to decrease because $n > P$. In this case, when a failure strikes an early task, the subsequent tasks could be assigned to other processors to reduce the impact of the failure. Ultimately, if $n \gg P$, MinExp would become more efficient. Indeed, since the tasks are almost independent and uni-processor tasks, list scheduling is able to dynamically balance the loads of different processors. Thus, minimizing the expected execution time of each individual task using MinExp would be a good strategy for the overall execution time of the workflow.

For most of the other workflows, we can similarly observe the same up-and-down effect as a result of these two phenomena, except for Sras, which is not impacted by the number of tasks. For this workflow, only a few key dominating tasks matter and their width remains well below the number of processors. Since these tasks form a small proportion of the total number of tasks, varying n does not significantly alter their chance of being hit by a failure, so the ratio remains close to 1.

6.5.4 Statistics

We provide some statistics related to the experiments of Section 6.5.3, still focusing on the four workflows Blast, Seismology, Genome and Sras. First, we check the quality of the strategy MinExp for each task, hence with N_{ME} segments: we make a comparison with the strategy Lambert that uses the exact optimal number of segments N_{opt} (with the notations of Section 6.2.5). In Table II, we report the mean and standard deviation of the ratio of the expected execution time achieved by Lambert over that achieved by MinExp. Hence, a value greater than 1 means that MinExp is better. Because we consider statistics

Table II: Geometric mean and standard deviation of the ratio of the expected execution time achieved by LAMBERT over that achieved by MINEXP.

	BLAST	GENOME	SEISMOLOGY	SRAS
Geometric Mean	1.00017	1.02691	1.02895	1.000041
Geometric SD	1.01564	1.02486	1.05626	1.00077
Instances where MINEXP is better	462(51.3%)	778(86.4%)	677(75.2%)	405(45%)
Instances where LAMBERT is better	436(48.4%)	122(13.6%)	220(24.4%)	444(49.3%)
Instances where both are identical	2(0.2%)	0(0%)	3(0.3%)	51(5.7%)
# tasks with identical number of checkpoints	95.5%	93.1%	94.2%	91.5%
# tasks where LAMBERT has one less checkpoint	4.5%	6.9%	5.8%	8.5%
# tasks where LAMBERT has neither of above	0.0%	0.0%	0.0%	0.0%

Table III: Average performance ratio of each checkpoint strategy.

	BLAST	GENOME	SEISMOLOGY	SRAS
MINEXP	1.19228	1.08755	1.19537	1.000737
LAMBERT	1.19256	1.11702	1.23072	1.000778
BASICCHECKMORE	1.02758	1.00688	1.00714	1.00329
CHECKMORE	1.02723	1.00698	1.00717	1.00330

on ratios, we use the geometric mean and standard deviation instead of classical values. For each workflow type, an instance in Table II corresponds to a given set of parameters (out of 30 possible sets), which is tested for 30 different task graphs. Hence there are 900 instances per workflow type. For each instance, the expected execution times are averaged over 50 failure scenarios.

In the last three rows of Table II, we compare the number of checkpoints for each task in each graph, with a total of around $25,000,000$ task comparisons per workflow type. The 0.0% is exactly 0 out of around 25 millions: we never found a task for which LAMBERT would not have either the same number of checkpoints as MINEXP, or one less checkpoint than MINEXP. Altogether, we conclude that MINEXP and LAMBERT perform almost the same. The slight superiority of MINEXP in terms of performance is due to its conservative approach: MINEXP rounds up the number of checkpoints of each task to the higher number (taking the ceiling instead of the floor), which turns out to be a good decision when several tasks execute in parallel.

Next in Table III, for each of the four workflow types, we report the average value, over all 900 instances, of the performance ratio of each checkpointing strategy. As in Section 6.5.2, the performance ratio is $\frac{T}{T_{base}}$, where T_{base} is the failure-free execution time of the workflow, and T is the execution time under the failure scenario. The major difference from the results of Section 6.5.3 is that we now add the LAMBERT strategy to the comparison. Clearly, MINEXP and LAMBERT are quite similar, while CHECKMORE and BASICCHECKMORE bring huge benefits, except for SRAS.

6.5.5 Summary

Our experimental evaluation demonstrates that MINEXP and LAMBERT are not resilient enough for checkpointing workflows, although they provide an optimal strategy for each individual task. However, CHECKMORE proves to be a very useful strategy, except for SRAS whose ratios are extremely low. When varying the key parameters, the simulation results nicely corroborate our theoretical analysis. Furthermore, the easy-to-implement

BasicCheckMore strategy always leads to ratios that are close to those of CheckMore, regardless of the parameters.

6.6 Related work

6.6.1 Scheduling Workflows

Scheduling a computational workflow consisting of a set of tasks in a dependency graph to minimize the overall execution time (or makespan) is a well-known NP-complete problem [66]. Only a few special cases are known to be solvable in polynomial time, such as when all tasks are of the same length and the dependency graph is a tree [84] or when there are only two processors [46]. For the general case, some branch-and-bound algorithms [83, 141] have been proposed to compute the optimal solution, but the problem remains tractable only for small instances. In the seminal work, Graham [71] showed that the list scheduling strategy, which organizes all tasks in a list and schedules the first ready task at the earliest time possible, achieves an execution time that is no worse than $2 - \frac{1}{P}$ times the optimum, where P denotes the total number of processors, i.e., the algorithm is a $(2 - \frac{1}{P})$-approximation. This performance guarantee holds regardless of the order of the tasks in the list. Some heuristics further explore the impact of different task orderings on the overall execution time, with typical examples including task execution times, bottom-levels and critical paths (see [108] for a comprehensive survey of the various heuristic strategies).

While the results above are for workflows with uni-processor tasks (or tasks that share the same degree of parallelism), scheduling workflows with parallel tasks has also been considered. Li [112] proved that, for precedence constrained tasks with fixed parallelism of different degrees (i.e., rigid tasks), the worst-case approximation ratio for list scheduling under a variety of task ordering rules is P. However, if all tasks require no more than qm processors, where $0 < q < 1$, the approximation ratio becomes $\frac{(2-q)P}{(1-q)P+1}$. Demirci et al. [48] proved an $O(\log n)$-approximation algorithm for this problem using divide-and-conquer, where n is the number of tasks in the workflow. Furthermore, for parallel tasks that can be executed using a variable number of processors at launch time (i.e., moldable tasks), list scheduling is shown to be an $O(1)$-approximation when coupled with a good processor allocation strategy under reasonable assumptions on the tasks' speedup profiles [60, 94, 110].

In this chapter, we augment the workflow scheduling problem with the checkpointing problem for its constituent tasks. We analyze the approximation ratios of some checkpointing strategies while relying on the ratios of existing scheduling algorithms to provide an overall performance guarantee for the combined problem.

6.6.2 Checkpointing Workflows

Checkpoint-restart is one of the most widely used strategy to deal with fail-stop errors. Several variants of this policy have been studied; see [80] for an overview. The natural strategy is to checkpoint periodically, and one must decide how often to checkpoint, i.e., derive the optimal checkpointing period. An optimal strategy is defined as a strategy that minimizes the expectation of the execution time of the application. For an preemptible application, given the checkpointing cost C and platform MTBF μ, the classical formula due to Young [173] and Daly [47] states that the optimal checkpointing period is $W_{YD} = \sqrt{2\mu C}$.

Going beyond preemptible applications, some works have studied task-based applications, using a model where checkpointing is only possible right after the completion of a task. The problem is then to determine which tasks should be checkpointed. This problem has been solved for linear workflows (where the task graph is a simple linear chain) by

Toueg and Babaoglu [158], using a dynamic programming algorithm. This algorithm was later extended in [21] to cope with both fail-stop and silent errors simultaneously. Another special case is that of a workflow whose dependence graph is arbitrary but whose tasks are parallel tasks that each executes on the whole platform. In other words, the tasks have to be serialized. The problem of ordering the tasks and placing checkpoints is proven NP-complete for simple join graphs in [7], which also introduces several heuristics. Finally, for general workflows, deciding which tasks to checkpoint has been shown #P-complete [76], but several heuristics are proposed in [77].

In this chapter, we depart from the above model [21, 76, 77, 158], and assume that each workflow task is a preemptible task that can be checkpointed at any instant. This assumption is quite natural for many applications, such as those involving dense linear algebra kernels or tensor operations. It is even mandatory for coarse-grain workflows: unless the failure rate can be decreased below the current standard, the successful completion of any large task, say executing a few hours with 1K nodes, is very unlikely.

6.7 Conclusion

In this chapter, we have investigated checkpointing strategies for parallel workflows, whose tasks are either sequential or parallel, and in the latter case either rigid or moldable. Because HPC tasks may have a large granularity, we assume that they can be checkpointed at any instant. Starting from a failure-free schedule, the natural MinExp strategy consists in checkpointing each task so as to minimize its expected execution time; hence MinExp builds upon the classical results of Young/Daly, and uses the optimal checkpointing period for each task. We derive a performance bound for MinExp, and exhibit an example where this bound is tight.

Intuitively, MinExp may perform badly in some cases, because there is an important risk that the delay of one single task will slow down the whole workflow. To mitigate this risk, we introduce CheckMore strategies that may checkpoint some tasks more often than other tasks, and more often than in the MinExp strategy. This comes in two flavors. CheckMore decides, for each task, how many checkpoints to take, building upon its degree of parallelism in the corresponding failure-free schedule. BasicCheckMore is just using, as degree of parallelism for each task, the maximum possible value $\min(n, P)$ (hence it is equivalent to CheckMore for independent tasks all running in parallel). The theoretical bounds for BasicCheckMore are not as good as those of CheckMore, but its performance in practice is very close, thus BasicCheckMore proves to be very efficient despite its simplicity.

An extensive set of simulations is conducted at large scale, using realistic synthetic workflows from WorkflowHub with between 8k and 70k tasks, and running on a platform with up to 50k processors. The results are impressive, with ratios very close to 1 on all workflows for both CheckMore strategies, while MinExp has much higher ratios, for instance 1.7 on average for Blast and 1.46 for Seismology. Hence, the simulations confirm that it is indeed necessary in practice to checkpoint workflow tasks more often than the classical Young/Daly strategy. As future work, we plan to extend the simulation campaign to parallel tasks (rigid or moldable), as soon as workflow benchmarks with parallel tasks are available to the community. We will also investigate the impact of the failure-free list schedule on the final performance in a failure-prone execution, both theoretically and experimentally. Indeed, list schedules that control the degree of parallelism in the execution may provide a good trade-off between efficiency (in a failure-free framework) and robustness (when many failures strike during execution).

Chapter 7

Revisiting I/O bandwidth-sharing strategies for HPC applications

In the last two chapters, we have studied checkpointing strategies for different scenarios. However, we have not considered bandwidth limitations when processing I/O operations to actually checkpoint, especially when several applications are checkpointing at the same time. To address this, we revisit in this chapter I/O bandwidth-sharing strategies for HPC applications. When several applications post concurrent I/O operations, well-known approaches include serializing these operations (FCFS) or fair-sharing the bandwidth across them (FairShare). Another recent approach, I/O-Sets, assigns priorities to the applications, which are classified into different sets based upon the average length of their iterations. We introduce several new bandwidth-sharing strategies, some of them simple greedy algorithms, and some of them more complicated to implement, and we compare them with existing ones. Our new strategies do not rely on any a-priori knowledge of the behavior of the applications, such as the length of work phases, the volume of I/O operations, or some expected periodicity. We introduce a rigorous framework, namely *steady-state windows*, which enables to derive bounds on the competitive ratio of all bandwidth-sharing strategies for three different objectives: minimum yield, platform utilization, and global efficiency. To the best of our knowledge, this work is the first to provide a quantitative assessment of the online competitiveness of any bandwidth-sharing strategy. This theory-oriented assessment is complemented by a comprehensive set of simulations, based upon both synthetic and realistic traces. The main conclusion is that our simple and low-complexity greedy strategies significantly outperform FCFS, FairShare and I/O-Sets, and we recommend that the I/O community implements them for further assessment. This chapter corresponds to Submission [S3] (see Chapter 9).

7.1 Introduction

HPC applications do not share computing resources: all the nodes assigned to a given application are dedicated to that application throughout its execution. Such a mode of operation is enforced to guarantee a sustained level of performance to all applications that execute concurrently on the platform. However, concurrent applications do share both the interconnexion network and the parallel file system. When several applications request to perform an I/O operation simultaneously, they have to share the resource, which leads to interferences and performance degradation.

Several researchers have already identified and addressed this problem (see [9, 52, 88, 155, 174, 176] among others). Performance degradation due to I/O is already significant for

m	number of applications
p_i	size (number of nodes) of application $\mathcal{A}_i$
r_i	release time of application $\mathcal{A}_i$
$W_i^{(j)}$	duration of work phase number j for application $\mathcal{A}_i$
$v_i^{(j)}$	volume of I/O operation number j for application $\mathcal{A}_i$
B	total bandwidth of the I/O system
b	bandwidth of each platform node
b_i	maximal bandwidth of application $\mathcal{A}_i$: $b_i = \min(p_i b, B)$
α_i^j	fraction of bandwidth assigned to $\mathcal{A}_i$ for I/O operation j (it can vary)
$[T_{begin}, T_{end}]$	steady-state window
$y_i(t)$	yield of application $\mathcal{A}_i$ at time t

Table I: Summary of main notations for Chapter 7.

current state-of-the-art platforms and is expected to worsen due to the faster increase in processing speed than in I/O bandwidth [130]. The problem can be partially mitigated by reducing the volume of data transfers, e.g., via compression or in-situ processing. But the main question remains: given several applications executing concurrently and competing for I/O resources, how to orchestrate I/O operations? In other words, scheduling strategies must be designed and evaluated to dynamically assign a fraction of the total I/O bandwidth to individual application transfers. Well-known strategies are FCFS, which gives exclusive I/O access to the first pending I/O operation, and FairShare, which assigns bandwidth proportionally to application transfers.

From a scheduling perspective, which fraction goes to which application at any given time depends upon the optimization metric, such as application progress rate (minimum or average) or platform utilization. When targeting fairness across concurrent applications, a classical objective is to maximize the minimum *yield*, where the yield of an application is the ratio of its actual progress rate over the progress rate that would have been achieved if the application was executing with a dedicated I/O system and always granted the total available bandwidth. We discuss optimization metrics in detail in Section 7.3.3.

This work focuses on I/O bandwidth-sharing scheduling strategies for HPC applications, revisiting existing strategies and introducing new ones. Our major contributions are described in the following four paragraphs.

General framework We provide and assess online scheduling strategies that are agnostic of the characteristics of the concurrent applications in terms of processing time and I/O requests. In particular, we do not assume any periodic behavior; several applications execute concurrently and alternate phases of work and phases of I/O operations, whose lengths are not known a priori. Instead, we discover the timing and size of I/O transfers on the fly, as each application posts its operations. We allow for interrupting and resuming on-going I/O operations dynamically, and launching newly posted ones.

Novel strategies We introduce novel I/O bandwidth-sharing strategies that aim at allocating a fraction of the bandwidth to each application as a function of the current progress of all applications. The main motivation is to maximize the minimum yield that can be achieved each time a scheduling decision is made. These novel heuristics come in several flavors, from simple greedy algorithms to sophisticated decision mechanisms.

Competitiveness analysis We provide a rigorous framework by focusing on a *steady-state* time window, defined with the following three rules. Throughout the window: (i) several applications, each with a processing history, execute concurrently; (ii) none of them terminates; and (iii) no new application can start. Thus, the window corresponds to a steady-state mode of behavior where each application progresses at the rate enforced by the I/O bandwidth-sharing strategy. Focusing on such a window is key to assess performance. Otherwise, say if some application would terminate before the end of the window, the batch scheduler would likely launch a new application, whose starting time and progress up to the end of the window would depend on all previous scheduling decisions. The same holds if a new application is launched in the middle of the window. Getting rid of the interaction with the batch scheduler, we provide the first complexity results on the performance of several I/O bandwidth-sharing strategies, some old and some new, and for various optimization objectives.

Comprehensive simulation campaign We compare existing and novel I/O bandwidth-sharing strategies on an extensive set of application scenarios, some generated from realistic traces derived from the APEX workflows report [109] , and some with synthetic parameters. A key parameter is the I/O pressure W, defined for a steady-state window $[T_{begin}, T_{end}]$, as the ratio $\frac{V}{B(T_{end}-T_{begin})}$, where (i) V is the total I/O volume (accumulated for all applications) to transfer during the window; and (ii) B is the total I/O bandwidth (see Section 7.6.1 for details). In a nutshell, if this ratio is close to 1 or even exceeds 1, the set of I/O operations saturate the I/O system, and many I/O operations will have to be delayed. We study how rapidly the performance of each strategy degrades for high I/O pressures, thereby paving the way for a fair bandwidth allocation on future platforms. We point out that simulations are a first but mandatory step to assess the limitations and strengths of all the I/O bandwidth-sharing strategies. Our extensive set of experiments corresponds to several months of platform usage and would have been impossible to deploy on a large-scale platform, even if we had both permission and budget to conduct them. The main conclusion is that two simple and low-complexity greedy strategies significantly outperform FCFS, FairShare and I/O-Sets, and we recommend that the I/O community would implement them for further assessment.

The chapter is organized as follows. We first survey related work in Section 7.2. Then, we detail the application and platform framework in Section 7.3, together with the optimization objectives. We detail well-known bandwidth-sharing strategies, and introduce new ones, in Section 7.4. Complexity results are stated in Section 7.5 in the form of lower bounds for competitive ratios. The experimental evaluation in Section 7.6 presents extensive simulation results comparing all the strategies. Finally, we conclude and provide hints for future work in Section 7.7.

7.2 Related Work

We discuss related work in this section. We survey existing approaches before pointing to a related problem in the scheduling literature.

CALCioM [52] This seminal paper introduces and experimentally compares three policies to manage cross-application coordination of I/O operations: (i) *Interference* (called FairShare in this chapter), where the total bandwidth is shared equally[1] among all

[1]More precisely, FairShare shares the total bandwidth in proportion to the size of the concurrent applications, see Section 7.4.1 for details.

concurrent operations; (ii) *FCFS-based serialization* (called FCFS in this chapter), where I/O operations are serialized based upon an FCFS priority; and (iii) *Interruption-based serialization*, where I/O operations are serialized but preemptive, allowing for another operation B to interrupt the current operation A, which resumes only after the completion of B. Some examples are given to explain when to favor a given strategy, but no general approach is explored. In particular, interruption-based serialization would require to set priorities among applications, which are not detailed in the chapter. Altogether, this work presents one of the first comparisons of bandwidth-sharing strategies, and we build upon their ideas to cast a general framework and introduce new strategies.

CLARISSE [88] This paper introduces a middleware designed to enhance data-staging coordination and control in the HPC software storage I/O stack. Among many other contributions, the CLARISSE middleware enables to directly compare the *no-scheduling* strategy (called FairShare in this chapter) with FCFS and reports performance gains for the latter. Intuitively, the superiority of FCFS can be expected as it comes from a classic result in parallel computing: when scheduling two identical communications that can each make use of the full bandwidth, better serialize them than execute them concurrently. Indeed, with serialization, the first communication ends at time t and the second one at time $2t$ (for a duration t, assuming a start at time 0), while in parallel, both communications end at time $2t$. However, our analysis and experiments reveal that this intuition can be misleading and that (i) FairShare prevails over FCFS in many practical scenarios and (ii) more sophisticated policies that account for past history to set priority-based bandwidth assignments perform even better.

I/O-Cop [155] I/O-Cop is a prototype system aimed at exploring access control mechanisms to manage the shared Parallel File System (PFS) of the platform. This work is motivated by revealing the contention incurred when several applications aim at performing I/O transfers simultaneously. The I/O-Cop prototype is limited to the case when the access controller to the Parallel File System (PFS) provides exclusive access to a single application at a given time, and without allowing for preemption of ongoing I/O operations.

QoS-based and reward-based approaches In [169], the authors also advocate controlling accesses to the PFS in order to achieve some Quality of Service (QoS) for each application. They envision a system with several I/O storage devices (disks, SSDs or NVRAMs) and aim at load-balancing I/O requests across all storage types to minimize contention. In [153], the authors consider several applications that execute concurrently and post I/O requests. They partition all the I/O requests into several queues, one per application, and aim at establishing priorities across the applications. The idea is that after completing some I/O transfer, a given application could be granted access for its next I/O transfer before all the other applications would have completed one I/O transfer themselves. At each time-step, the progress of each application is monitored as the number of I/O transfers that have been granted so far. In a related paper [85], the authors survey I/O capabilities of state-of-the-art supercomputers and enforce QoS constraints for I/O transfers by implementing a token-based bucket algorithm that works similarly to that of [153]. Finally, the authors of [132] target a system with several I/O sub-systems (OST, which stands for Object Storage Target, typically a RAID array of disks). For each application, they aim at the same share of available bandwidth on each OST, because it balances transfers (one needs to wait for the last node to complete its transfer before resuming work). The allocation of nodes (hence applications) to the different OSTs is given by some external mechanism.

Then, on a given OST, some application may benefit from an increased bandwidth, which is done by throttling another application. The throttled application is issued a coupon, to be redeemed later. They do not deal with the interplay of successive I/O operations and work phases, and no comparison is made with other strategies. In contrast, our work restricts to a single OST but provides a comprehensive comparison of several bandwidth-sharing strategies

Periodic applications A series of papers [8,9,38,65,95] focus on periodic applications that consist of work phases followed by I/O operations. More precisely, each application repeats a two-phase period with a fixed computing length followed by an I/O of volume. The CPU lengths and I/O volume depend upon the application, but remain the same from one period to the next. The major goal of these works is to orchestrate a global periodic scheme where I/O transfers are meticulously shaped to fill up the smallest possible rectangle that will repeat. While the problem of finding the minimum size rectangle is shown to be NP-complete in the initial work [65], several interesting heuristics have been developed in the subsequent papers. The approach is quite flexible, with I/O transfers possibly split into different sub-transfers, each with a different bandwidth. The main limitation is of course the assumed periodicity of each application. An extension is provided by other authors in [176], where applications still consist of phases with work followed by I/O transfers, but now CPU phases have stochastic lengths taken from some probability distribution, while I/O phases have constant length. As a motivation, for CPU phases, we can think of a constant amount of flops to perform, with some system-dependent or data-dependent noise, while for I/O transfers, we can think of a fixed-size checkpoint operation. In contrast, our approach does not assume any a priori knowledge of the concurrent applications.

I/O-Sets [174] This recent work can be viewed as an interesting extension of the work in [9] for periodic applications. Each application consists of several iterations, which as above are work phases followed by I/O operations. Periodicity is no longer assumed. Instead, for each application, they determine the value of ω, which is the average length of an iteration so far. In [174], CPU lengths and I/O volumes are sampled from some probability distributions (that differ for each application), which enables to compute ω with the expectations of these distributions, but one could envision to acquire the value of ω on the fly, as the application progresses. Then, the applications are partitioned into I/O-sets: two applications belong to the same set if they have the same value for $\lceil \log_{10} \omega \rceil$. Each I/O-set is assigned a priority. The I/O bandwidth-sharing strategy is described in detail in Section 7.4.2. In a nutshell, FCFS is enforced within each I/O-set; hence, at most one application per I/O-set is competing for bandwidth at any time-step. Then some priority-based sharing is enforced across I/O-sets. The motivation for using a mixture of FCFS and FairShare (or more precisely a priority-based variant of sharing) is very interesting: small and large applications (characterized by different orders of magnitude for ω) should not be treated equally by the scheduler. The I/O-sets strategy has several parameters, and we use the same instantiation as in [9], with the same name Set-10. We use Set-10 as a competitor for our novel strategies.

A note on the painter problem In the scheduling literature, the *painter* problem, a.k.a the *scheduling with delays* problem, is the following: (i) several chains of tasks are to be scheduled on a single machine; (ii) for each chain, there is a minimal *delay* to be enforced between the completion of a task and the start of its successor. As for the analogy with a painter: the painter is the machine and has several rooms to paint on its agenda, each

with several paint layers (a task is the application of a paint layer); for each room (each chain), there is a delay between the end of a layer and the next one. The tasks are not preemptive. Release times can be simulated by adding delays from a fake source task. This is an offline problem where the objective is to minimize either the makespan (maximum completion time of a task) or the total flow (unweighted or weighted sum of all completion times). The analogy with the I/O problem is clear: the machine is the I/O resource, the task chains are the applications, the tasks are the I/O operations, and the delays are the computing phases between two consecutive I/O operations. The main differences with the I/O scheduling problem are the following:

1. Execution is not preemptive in the painter problem, while one can pause an on-going I/O operation;
2. A single task is executed at any time step while several I/O operations (from different applications) can share the I/O bandwidth;
3. All chain parameters (task lengths and delay values) are known at the beginning of the execution while the lengths of work phases and the volumes of I/O operations are discovered on the fly in the I/O scheduling problem.

Particular instances of the painter problem have been shown to have polynomial complexity. We refer to the interested reader to [32, 55, 115, 125, 126] for details. A survey of recent results and extensions is available in [107].

7.3 Framework

In this section, we describe the framework. We start with application characteristics and detail rules for I/O operations and bandwidth allocation in Section 7.3.1. We discuss the interaction with the batch scheduler and explain why we restrict to steady-state time windows in Section 7.3.2. We conclude with optimization objectives in Section 7.3.3. Main notations are summarized in Table I.

7.3.1 Applications

Application Characteristics

We consider a very general framework where applications are submitted online to the batch scheduler. Each application $\mathcal{A}_i$ requests p_i nodes and starts executing as soon as the batch scheduler has been able to allocate that many nodes. Thus, each application executes on a dedicated set of nodes throughout its execution, which is the standard approach on large-scale HPC platforms. However, all applications execute I/O transfers (reads and writes) through the I/O controller and share the bandwidth of the I/O system. Our approach is agnostic of the nature of the storage (SSDs, NVRAMs, disks or tapes), and of the organization of the PFS (Parallel File System).

Each application $\mathcal{A}_i$ executes an alternating sequence of work phases and I/O operations, which we represent as follows:

$$\mathcal{A}_i \equiv \quad v_i^{(0)}, W_i^{(1)}, v_i^{(1)}, W_i^{(2)}, \ldots, v_i^{(n_i-1)}, W_i^{(n_i)}, v_i^{(n_i)}, \ldots$$

where $v_i^{(j)}$ stands for I/O volumes, and $W_i^{(j)} > 0$ stands for (parallel) work units. Because computing nodes are dedicated to the application, we can assume w.l.o.g. that one unit of work lasts one second, so that the $W_i^{(j)}$ represent the duration of the work phases; more precisely, within W_i seconds, each of the p_i nodes performs W_i work units. However, because we do not know the bandwidth of I/O operations in advance, we have to express

them in volume (amount of bytes) rather than in duration. We detail rules for bandwidth allocation in Section 7.3.1. We will discuss rules for posting and managing I/O operations in Section 7.3.2.

As stated before, the length $W_i^{(j)}$ of each work phase is not known until it terminates, and the volume $v_i^{(j)}$ of each I/O operation is not known until the operation is posted to the I/O controller. Similarly to the related work surveyed in Section 7.2, we also assume that I/O operations are blocking and coordinated between the different nodes of the application, and that the application does not overlap I/O operations with some work phase. This is typical of HPC applications using a synchronous global interface like MPI-IO [100, 120], which also provides the I/O controller with critical information like the volume of data to transfer.

Bandwidth Allocation

Consider an application $\mathcal{A}_i$ executing on p_i nodes and initiating an I/O operation of volume $v_i^{(j)}$. What are the bandwidth allocation rules for this operation? We let b be the bandwidth of the network card (of interface card) of each node, and B be the total bandwidth of the I/O system.

First, assume for simplicity that the I/O operation is not interrupted, and is granted the same bandwidth from start to completion. The maximal bandwidth that can be granted by the I/O controller is

$$b_i = \min\left(p_i b, B\right). \tag{7.1}$$

Note that Equation (7.1) implicitly assumes that each node of $\mathcal{A}_i$ has to transfer (approximately) the same volume of data to/from the PFS. If transfers are unbalanced from one node to another, we should redefine $v_i^{(j)}$ as $v_i^{(j)} = p_i v_{i,\max}^{(j)}$, where $v_{i,\max}^{(j)}$ is the maximum volume of data to be transferred by any of the p_i nodes of $\mathcal{A}_i$. The main rule of the game for the scheduler is to assign a fraction $\alpha_i^{(j)}$ of the maximal bandwidth b_i to the I/O operation $v_i^{(j)}$. The duration of the I/O operation will then be

$$d_i^{(j)} = \frac{v_i^{(j)}}{\alpha_i^{(j)} b_i}. \tag{7.2}$$

Of course, if no I/O operation has been posted by another application, the scheduler will enforce $\alpha_i^{(j)} = 1$ to ensure fastest possible completion. In that case, we use the notation

$$d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i} \tag{7.3}$$

to denote the minimal possible duration of the I/O operation. On the contrary, in the presence of several concurrent I/O operations, the scheduler will resort to some bandwidth-sharing strategies, like the ones studied in this chapter.

We are ready to discuss the general case, which will require some additional notations. Intuitively, a given I/O operation will NOT be granted the same bandwidth fraction throughout execution. At any time-step t, some I/O operations that were posted before are granted some bandwidth and executing, while some others may be pending (that is to say their fraction is currently 0). A new I/O operation may be posted at time t, which the scheduler can account for by granting it some bandwidth, at the price of reducing the fraction of other applications. On the contrary, some on-going I/O operation may complete at time t, thereby opening the possibility of a larger fraction to be granted to some

applications. We see that bandwidth fractions are granted only for some duration, which we call the *horizon*. Decisions are taken at specific instants, which we call *events*. Typically, an event corresponds to the posting of a new I/O operation, or to the termination of an on-going one. But an event can also be triggered by the I/O scheduler, e.g., for a strategy where additional events are created periodically, say every 10 seconds. The I/O controller takes a new decision at every event, as explained below. The constraints on the number of events, and the cost of bandwidth-sharing strategies, will be detailed in Section 7.3.2.

Consider an event at time t, and let $S(t)$ be the index set of *active* applications, i.e., applications that have posted an I/O operation before time t which is not yet completed, or applications that post a new I/O operation exactly at time t. Among the applications with incomplete I/O-operations, some may be transferring data at some bandwidth fraction and some may be kept waiting. Each active application $\mathcal{A}_i$, $i \in S(t)$, is allotted a bandwidth $\alpha_i^t b_i$ (with some α_i^t possibly 0) so that

$$\sum_{i \in S(t)} \alpha_i^t b_i \leq B. \tag{7.4}$$

This bandwidth allocation remains valid until the next event at time $t + h$, where h is the horizon. The bandwidth allocation depends upon the bandwidth-sharing strategy, whose inputs are the volume of data that must still be transferred for each on-going I/O operation, the knowledge of the progress of all active applications so far, and the optimization objective.

We stress that the horizon h is unknown at time t. The next event is triggered either by a new post or a completion, or again by an external decision given to the I/O controller. At time t, after having granted bandwidth fractions to active applications, we only know that h is greater than the time needed to complete the shortest on-going I/O operation, given that no new event (new post or external) will happen before that.

When the next event takes place at time $t + h$, we update the set of active applications, leaving out I/O operations that have completed and including new posts, if any. We also update the remaining volume of data still to be transferred for each active application. The I/O controller applies the bandwidth-sharing strategy for this new set of parameters.

7.3.2 Steady-State Windows

In this section, we recall the management of HPC applications by the batch scheduler, and explain why we need to restrict to steady-state time windows to assess the performance of bandwidth-sharing strategies.

Interaction with the Batch Scheduler

HPC applications are submitted to the batch scheduler. Each application $\mathcal{A}_i$ has a release time r_i, a size p_i and a wall-time res_i (length of the reservation slot). Upon release, application $\mathcal{A}_i$ is put in the queue of the batch scheduler and will be allocated resources at time $t_i^{alloc} \geq r_i$, which means that p_i nodes are dedicated to the application during the interval $[t_i^{alloc}, t_i^{alloc} + res_i)$. The p_i nodes are released as soon as the application completes its execution or its deadline is reached, whichever comes first.

Each application has dedicated nodes but all applications that execute concurrently share the I/O system. I/O operations are posted by the applications and managed by the I/O controller. If an application posts an I/O operation while another I/O operation has already been granted access, several scenarios can happen, depending upon the bandwidth-sharing policy implemented by the I/O controller. We have already discussed the FCFS

and FairShare strategies in Section 7.2, and will introduce other strategies in Section 7.4. Whenever the I/O controller makes a decision according to its bandwidth-sharing policy, this decision has an impact on the progress of all active applications. Altogether, the bandwidth-sharing policy will change the termination time of all applications. In theory, some applications may even fail to complete before the end of their reservation due to the bandwidth-sharing strategy being disadvantageous to them. On the contrary, some applications may benefit from the strategy and complete early, thereby releasing their resources early. In summary, the opportunities for decisions of the batch scheduler to allocate new applications will depend upon the bandwidth-sharing strategy applied to the applications that are currently executing. Furthermore, any decision of the batch scheduler changes the mix of applications that run concurrently and possibly compete for I/O resources. This, in turn, changes the scope and impact of the decisions of the bandwidth-sharing policy implemented by the I/O controller. Altogether, the interplay between the batch scheduler and the decisions of the I/O controller is hard to comprehend.

To the best of our knowledge, none of the papers surveyed in Section 7.2 has dealt with this difficulty. Instead, these papers consider a fixed number of applications that execute concurrently (each on a dedicated set of nodes) and compete for I/O access. This amounts to consider an execution window $[T_{begin}, T_{end}]$ where all applications start executing at time T_{begin} and do not complete execution before time T_{end}, regardless of the I/O policy that is implemented. In other words, the platform operates in steady-state mode during the window $[T_{begin}, T_{end}]$ with no application terminating nor no new application launched throughout the window. This assumption is never stated in recent papers. Again, the reason why it is assumed that applications do not complete before the end of the window is the following: if an application terminates at time $T < T_{end}$, the batch scheduler might launch another application right after the completion. Because T depends on the bandwidth-sharing strategy that is enforced, it becomes impossible to assess the performance of the strategy by itself.

In this chapter, we use a steady-state execution window $[T_{begin}, T_{end}]$ and assume that m applications $\mathcal{A}_i$ $(1 \leq i \leq m)$ execute concurrently throughout the window. To eliminate side effects and deal with a general scenario, we do not assume that the applications start executing at time T_{begin}: on the contrary, the applications may have been launched earlier and have been executing for some time. The history of the applications will be taken into account when evaluating the objective function (see Section 7.3.3).

Cost Model for Steady-State Windows

Given a steady-state execution window $[T_{begin}, T_{end}]$, assume that m applications $\mathcal{A}_i$ $(1 \leq i \leq m)$ execute concurrently throughout the window. Each application $\mathcal{A}_i$ will execute a series of work phases followed by I/O transfers. If the application $\mathcal{A}_i$ was alone on the platform, all I/O transfers would be granted maximal bandwidth b_i. Let $N_{op}(i)$ be the number of I/O operations that would be initiated from time T_{begin}until time T_{end}, assuming such a dedicated mode.

In concurrent mode, we introduce two *events* for each I/O operation, one when it is posted, and one when it completes. The total number of events due to I/O operations is upper bounded by

$$E = \sum_{i=1}^{m} 2N_{op}(i). \tag{7.5}$$

Indeed, no application will perform more I/O operations by the end of the window than in dedicated mode, hence the number of events for each application $\mathcal{A}_i$ never exceeds $2N_{op}(i)$,

regardless of the bandwidth-sharing strategy.

The value of E is a key parameter to the size of the problem (other parameters include the binary encoding of work lengths and I/O volumes). We enforce that all bandwidth strategies have a cost polynomial in E, meaning that the number of bandwidth-sharing decisions remains polynomial in E. For instance, if the I/O controller enforces periodic decisions every h seconds, where h is a fixed horizon, the number of additional events $E^{(+)} = \lfloor \frac{T_{end} - T_{begin}}{h} \rfloor$ must remain polynomial in E. We use $E^{(+)} = E$ in the simulations to add equi-spaced decisions across the steady-state window. Note that triggering an external event every second would lead to $T_{end} - T_{begin}$ external events, which is exponential in the problem size (we use a logarithmic encoding for all parameters).

To the best of our knowledge, none of the papers surveyed in Section 7.2 has discussed how frequently decisions should be taken, nor has included the cost of the bandwidth-sharing strategy each time a decision is taken. We could easily include that cost into the assessment of the performance of the strategies. We do not, because the cost is inherent to the strategy and independent of the actual length of the work phase and I/O operations: if we multiply the latter quantities (and the window size) by a factor 10 or 100, the cost of the strategy remains the same and becomes negligible in front of the execution time of the applications.

7.3.3 Objectives

In this section, we define the *yield* of an application. The major objective of our novel bandwidth-sharing strategies is MinYield, the maximization of the minimum yield over all applications executing within the steady-state window $[T_{begin}, T_{end}]$. However, we also report performance for two other objectives, Utilization and Efficiency, which we describe at the end of this section.

Consider an application $\mathcal{A}_i$ that is released at time $r_i = 0$. Consider a steady-state window $[T_{begin}, T_{end}]$. At any time $t \geq T_{begin}$, we want to monitor the progress of $\mathcal{A}_i$ in terms of work done and data volume transferred. Recall that $\mathcal{A}_i$ executes an alternating sequence of work phases (work) and I/O operations:

$$\mathcal{A}_i \equiv \quad v_i^{(0)}, W_i^{(1)}, v_i^{(1)}, W_i^{(2)}, \ldots, v_i^{(n_i-1)}, W_i^{(n_i)}, v_i^{(n_i)}, \ldots$$

We have assumed unit speed for work phases, and we normalize I/O volumes by the maximal possible bandwidth $b_i = \min(p_i b, B)$. Letting $d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i}$ be the minimum duration for I/O operation number j of volume $v_i^{(j)}$, we rewrite $\mathcal{A}_i$ as

$$\mathcal{A}_i \equiv \quad d_{i,\min}^{(0)}, W_i^{(1)}, d_{i,\min}^{(1)}, W_i^{(2)}, \ldots, d_{i,\min}^{(n_i-1)}, W_i^{(n_i)}, d_{i,\min}^{(n_i)}, \ldots$$

The *ideal progress* of $\mathcal{A}_i$ at time t is the amount of work plus the volume of data transferred since its release time r_i and up to time t, when all I/O operations have taken place with no delay and at the maximal possible bandwidth b_i. This corresponds to $\mathcal{A}_i$ progressing at maximal rate, which happens if it executes in dedicated mode on the platform. By definition, at time t, the ideal progress is equal to $t - r_i$.

In a concurrent execution, the *actual progress* of $\mathcal{A}_i$ at time t is the amount of work plus the volume of data transferred since its release time r_i and up to time t. While work phases still progress at full (unit) speed, I/O operations are slowed down by interferences. For any time $t \in [T_{begin}, T_{end}]$, let $W_i^{(done)}(t)$ be the total amount of work done up to time t, and $V_i^{(transferred)}(t)$ be the total volume of data transferred up to time t. The *yield* of $\mathcal{A}_i$

at time t is defined as the ratio of the actual progress over the ideal progress, namely

$$y_i(t) = \frac{W_i^{(done)}(t) + \frac{V_i^{(transferred)}(t)}{b_i}}{t - r_i}. \tag{7.6}$$

As a side note, we show how to compute the value of $V_i^{(transferred)}(t)$ as the concurrent execution goes. We do this computation incrementally, one work phase or I/O operation after another. Consider the I/O operation number j and assume that it has occurred during the interval $[start_i^{(j)}, end_i^{(j)}]$ ($end_i^{(j)}$ is equal to the completion time of this I/O operation and $start_i^{(j)}$ to the completion time of the previous work phase). Let $\alpha_i^{(j)}(u)b_i$ be the bandwidth granted at time $u \in [start_i^{(j)}, end_i^{(j)}]$, where $0 \leq \alpha_i^{(j)}(u) \leq 1$ (and let $\alpha_i^{(j)}(u) = 0$ for u outside this interval). If the I/O operation number j is not complete at time t, i.e., if $t \in [start_i^{(j)}, end_i^{(j)})$, the amount of data volume $V_i^{(j)}(t)$ transferred up to time t is

$$\int_{start_i^{(j)}}^{t} \alpha_i^{(j)}(u) b_i du = V_i^{(j)}(t). \tag{7.7}$$

In fact, the integral is a discrete sum of at most E components, since we change bandwidth allocation only when a new event takes place. Note that if $t \geq end_i^{(j)}$, we obtain $V_i^{(j)}(t) = v_i^{(j)}$. Equation (7.7) enables us to compute the actual progress incrementally, from one work phase or I/O operation to the next. Of course, the actual progress depends upon the bandwidth-sharing strategy through the choice of the fractions $\alpha_i^{(j)}(u)$ of the maximal bandwidth b_i allotted at every instant u.

We are ready to state the optimization objectives, together with their initial motivation. Consider a steady-state window $[T_{begin}, T_{end}]$ and m applications. Each application $\mathcal{A}_i$ has a yield $y_i(T_{begin})$ when entering the window. The three target objectives are MinYield, Utilization and Efficiency.

MinYield The objective is to maximize the minimum yield at the end of the window:

$$\text{Maximize} \min_{1 \leq i \leq m} y_i(T_{end}). \tag{7.8}$$

This objective aims at enforcing fairness among all the applications, regardless of their characteristics. The intuition is that all applications suffer from the same slowdown factor if they achieve the same yield. As discussed in Sections 7.1 and 7.2, previous work has shown the limitations of FCFS and FairShare, which give priority to some applications and severely slow down other ones. MinYield will guide bandwidth-sharing decisions so that all applications exit the window with balanced yields. An application entering the window with a very low yield will be granted more bandwidth to catch up.

Utilization The objective is to maximize platform utilization throughout the window:

$$\text{Maximize} \frac{\sum_{1 \leq i}^{m} p_i \left(W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin}) \right)}{(T_{end} - T_{begin}) \sum_{1 \leq i}^{m} p_i}. \tag{7.9}$$

The work $W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin})$ done by each application $\mathcal{A}_i$ within the window is weighted by its size p_i. This objective is the classical performance objective from the perspective of the administrator or owner of the platform, because it measures the fraction of time where computing nodes have been used for actual application work. Hence, this

objective is natural for HPC applications that perform no or little I/O transfers. However, it may seem ill-suited in a framework focusing on I/O transfers, because it is very sensitive to the ratio of work over data volumes (normalized by maximal bandwidth). For instance, if we multiply all data volumes by, say, 10, platform utilization will plummet, even if we keep the same bandwidth-sharing strategy. This observation leads to introducing the objective EFFICIENCY.

Efficiency The objective is to maximize the sum of the actual progress of all applications throughout the window:

$$\text{MAXIMIZE} \frac{\sum_{1\leq i}^{m} p_i \left(W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin}) + \frac{V_i^{(transferred)}(T_{end}) - V_i^{(transferred)}(T_{begin})}{b_i} \right)}{(T_{end} - T_{begin}) \sum_{1\leq i}^{m} p_i} \tag{7.10}$$

Comparing Equations (7.9) and (7.10), we see that I/O operations are taken into account with EFFICIENCY: this objective aims at optimizing the combined progress of all applications. It can be viewed as a measure of how efficiently platform resources (both compute nodes and the I/O system) are used.

7.4 Bandwidth-Sharing Strategies

We describe bandwidth-sharing strategies in this section. We start by recalling a few notations and introducing new ones. Consider a steady-state window $[T_{begin}, T_{end}]$ with m applications executing concurrently. Consider an event at time t and let $S(t)$ be the index set of active applications at time t. Note that applications that are not active are engaged in work phases at time t and progress independently of the decisions made by the I/O controller.

Each active application $\mathcal{A}_i$, $i \in S(t)$, has posted an I/O operation at time $R_i \leq t$ that is not complete at time t. Let $\mathcal{V}_i$ denote the remaining volume still to be transferred for the I/O operation. Each active application is allotted a fraction α_i^t (with some α_i^t possibly 0) of its maximum possible bandwidth $b_i = \min(p_i b, B)$. The bandwidth-sharing strategy consists in determining α_i^t for each active application $\mathcal{A}_i$. Finally, let $\mathcal{BW}_i(t', y)$ denote the bandwidth that should be alloted to application $\mathcal{A}_i$ for it to achieve a yield of at least y at time t'.

We start with some simple greedy strategies, some old and some new, in Section 7.4.1. Then in Section 7.4.2, we detail the recent SET-10 strategy proposed in [30]. Finally, in Section 7.4.3, we sketch an elaborate strategy whose aim is to compute the best horizon for maximizing the minimum yield.

7.4.1 Greedy Strategies

We discuss below six greedy strategies. The first three strategies do not rely on any (tentative) horizon, while the last two aim at taking some future events into account. Finally, the sixth strategy re-evaluates the current bandwidth allocation at periodic time-steps.

- FAIRSHARE: each active application $\mathcal{A}_i$ with $i \in S(t)$ is allocated the bandwidth $\alpha_i = \min(1, \frac{B}{\sum_{j\in S(t)} b_j})$. Therefore, each application will either saturate its maximal bandwidth b_i, or it will receive a fair share (proportional to its size p_i) of the total bandwidth B. This is the de-facto strategy implemented by the parallel filesystems

available in most HPC centers. This strategy does not need to consider what application is requesting the I/O operation, but just how many I/O operations are currently concurrent.

- FCFS: greedily allocate the bandwidth to active applications by non-decreasing order of arrival R_i. More precisely, up to some re-ordering, let $S(t) = \{1, 2, \dots k\}$ with $R_i \leq R_{i+1}$ for $1 \leq i < k$. $\mathcal{A}_1$ is granted its maximum bandwidth b_1 (hence, $\alpha_1 = 1$), then $\mathcal{A}_2$ is granted $\alpha_2 b_2 = \min(b_2, B - \alpha_1 b_1)$, and so on until no more bandwidth is available.
- GreedyYield: greedily allocate the bandwidth to active applications sorted by non-decreasing yields $y_i(t)$. The greedy allocation process is the same as for FCFS but with a different criterion, current minimum yield instead of oldest posting time. This strategy gives priority to applications with low yield, so that they can catch up.
- GreedyCom: greedily allocate the bandwidth to the applications sorted by non-decreasing ratio $\mathcal{V}_i/b_i$, i.e., by the remaining time to complete the pending I/O operation at maximum possible bandwidth. This strategy gives priority to completing shorter transfers, with the goal of freeing the I/O system as fast as possible and/or give more bandwidth to forthcoming I/O operations.
- LookAheadGreedyYield: for each active application $\mathcal{A}_i$, compute the minimum yield Z_i that can be achieved (over all active applications) if $\mathcal{A}_i$ is given priority and allocated the maximum possible bandwidth b_i, and where the remaining bandwidth $B - b_i$ is allocated following GreedyYield for the other applications in $S(t)$. Then, we retain the allocation that maximizes the minimum yield Z_i obtained with these $|S(t)|$ possible priority choices. The rationale for LookAheadGreedyYield is to look ahead and maximize the minimum yield not at time t, but at time $t + h$, where the horizon h is (tentatively) computed as the end of one ongoing I/O operation.
- PeriodicGreedyYield (δ): this strategy is a variant of GreedyYield where I/O decisions are triggered by external (periodic) events submitted to the I/O controller every δ seconds, in addition to the regular events that correspond to posting and completion of I/O operations. As discussed in Section 7.3.2, we must restrict to a polynomial number of external events. With the notations of Section 7.3.2, we use $E^{(+)} = E$ in the simulations, which leads to choosing $\delta = \frac{T_{end} - T_{begin}}{E^{(+)}}$. At every event, external or regular, bandwidth-sharing decisions are the same as for GreedyYield. The rationale for adding periodic events is to avoid the risk that GreedyYield would apply a bad decision for too long: with several concurrent I/O operations lasting for a long time, greedy decisions are updated every δ seconds, instead of waiting for the first completion of one of these I/O operations.

7.4.2 Set-10 Strategy

This section provides a description of Set-10, the I/O-sets bandwidth-sharing strategy from [30].

Determination of I/O-sets With the notations of Section 7.3.3, consider an application $\mathcal{A}_i$ composed of operations

$$v_i^{(0)}, W_i^{(1)}, v_i^{(1)}, W_i^{(2)}, \dots, v_i^{(n_i-1)}, W_i^{(n_i)}, v_i^{(n_i)}, \dots$$

Assume that $\mathcal{A}_i$ has just completed the I/O operation $v_i^{(j)}$. Then, the current value of ω^i, the average length of an iteration for $\mathcal{A}_i$, is defined as

$$\omega^i = \frac{1}{j}\sum_{k=1}^{j}(W_i^{(k)} + d_{i,\min}^{(k)}),$$

where $d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i}$, and $b_i = \min(p_i b, B)$. Note that we neglect the initial I/O operation $v_i^{(0)}$ to match the specification of [30]. Then, $\mathcal{A}_i$ is assigned to I/O-set $\mathcal{S}_n$, where $n = \lfloor \log_{10} \omega^i \rceil$, and $\lfloor x \rceil$ denotes the nearest integer to x. Note that an application $\mathcal{A}_i$ may be dynamically reassigned to another I/O-set depending upon the duration of its next work phases and I/O operations. In [30], I/O-set $\mathcal{S}_n$, where $n = \lfloor \log_{10} \omega^i \rceil$, receives a priority $q_n = 10^{-n}$.

Bandwidth assignment Consider an event occurring at time t, and let $S(t)$ denote the index set of active applications that have a pending I/O transfer at time t. Each participating application $\mathcal{A}_i$, $i \in S(t)$, is allotted a bandwidth $\alpha_i b_i$ computed via the following algorithm [30]:

1. Assume that the applications in $S(t)$ belong to s different I/O-sets $\mathcal{S}_{n_1}, \mathcal{S}_{n_2}, \ldots, \mathcal{S}_{n_s}$.
2. Within each I/O-set, a single application is granted access to the I/O system. In other words, there is exclusive access within sets. If several applications in $S(t)$ belong to the same I/O set, the one with the smallest value of R_i (FCFS, the one that posted its request first) is selected.
3. Now, we have a subset of s applications, one per I/O subset, which will be granted some bandwidth. The intuition is to partition the bandwidth according to the priorities defined above. For simplicity, let us renumber the applications so that $\mathcal{A}_j$ is the application chosen from set $\mathcal{S}_{n_j}$, for $1 \le j \le s$. Then, each application $\mathcal{A}_j$ should be granted the fraction $\alpha_j = \frac{q_{n_j}}{\sum_{1 \le k \le s} q_{n_k}}$ of the total bandwidth B.
4. As usual, this bandwidth assignment remains valid until the next event.

However, this bandwidth-sharing algorithm implicitly assumes that each application can use the whole system bandwidth: $b_i = B$ for each application $\mathcal{A}_i$. To cope with general scenarios where this is not the case, we have to extend the algorithm. The natural idea is allocate bandwidth to several applications in the same I/O subset, rather than one, while still enforcing the priorities. More precisely, the fraction $\frac{q_{n_j}}{\sum_{1 \le k \le s} q_{n_k}}$ of the total bandwidth B is now assigned to several applications from $\mathcal{S}_{n_j}$, chosen greedily in FCFS order. Here is the extended algorithm for bandwidth-sharing:

1. Assume that the applications in $S(t)$ belong to s different I/O-sets $\mathcal{S}_{n_1}, \mathcal{S}_{n_2}, \ldots, \mathcal{S}_{n_s}$.
2. For each I/O-set $\mathcal{S}_{n_j}$, compute the maximum bandwidth fraction that it can receive, namely $\beta_j = \frac{\sum_{k \in \mathcal{S}_{n_j}} b_k}{B}$. As before, let $\alpha_j = \frac{q_{n_j}}{\sum_{1 \le k \le s} q_{n_k}}$.
3. We partition the s I/O sets into two categories, those that can receive the fraction α_j and those that are limited by their maximal bandwidth fraction β_j. Let $\mathcal{C}$ be the set of I/O sets of the latter category, i.e., such that $\beta_j \le \alpha_j$.
4. All the applications $\mathcal{A}_k$ in an I/O set belonging to $\mathcal{C}$ receive their maximal bandwidth b_k.

5. We compute the remaining bandwidth $B_{left} = (1 - \sum_{\mathcal{S}_{n_j} \in \mathcal{C}} \beta_j)B$.

6. We repeat the whole procedure with the remaining I/O-sets and B_{left}, until either there is no I/O-set left, or all remaining I/O-sets have a larger maximal bandwidth than their priority share: $\beta_j \geq \alpha_j$. In the final step, the remaining I/O-sets are granted the fraction α_j of the remaining bandwidth B_{left}. Within each of these I/O sets, bandwidth is allotted greedily in FCFS order.

Remark on Framework The I/O-sets strategy [30] does not assume that the total volume of an I/O operation is known when that operation is posted. Instead, they assume that this volume is unknown until the I/O operation ends. They rely on the knowledge of the average length of an iteration for each application, which is acquired from past behavior traces. In our simulations of SET-10, we acquire information on average iteration length on the fly as execution progresses.

As stated in Section 7.3.1, we do assume that the total volume of each I/O operation is known when posted. This knowledge is necessary for GREEDYCOM, LOOKAHEADGREEDYYIELD (described in Section 7.4.1) and BESTNEXTEVENT (described below in Section 7.4.3). However, GREEDYYIELD and LOOKAHEADGREEDYYIELD (also described in Section 7.4.1) do not need any information at all on the applications, they only need to compute application yields on the fly. And of course FAIRSHARE and FCFS do not need any information either.

7.4.3 Maximizing the Minimum Yield at the Next Event

Given an event at time $t \in [T_{begin}, T_{end}]$, the aim of strategy BESTNEXTEVENT is to find the *best predictable* event in the remainder of the window $]t, T_{end}]$. A predictable event is either the end of the execution window (at time T_{end}) or the first time one of the currently on-going I/O operations is completed, whichever comes first. The best predictable event is the predictable event at which point the minimum yield will be maximized. Of course, if an unpredictable event, such as the posting of a new I/O operation, surges before the best predictable event, the bandwidth-sharing strategy will account for it and recompute the best predictable event from that time on.

A priori, there are infinitely many dates in the interval $[t, T_{end}]$, at which the next predictable event can happen; hence, we cannot test each and every one of them. Instead, we partition the interval $[t, T_{end}]$ into a polynomial (in practice, quadratic) number of sub-intervals. The extremities of these sub-intervals will be either the earliest date at which an I/O operation can complete, or the time at which the characteristic yield functions of two applications intersect (see below for details; the characteristic yield function of an application will be, for instance, its maximum achievable yield at time t', or its yield at time t' if it is allocated no bandwidth, etc.).

Let $t = t_1 \leq t_2 \leq \ldots \leq t_{n_{\text{int}}} = T_{end}$ be the extremities of these sub-intervals. For each sub-interval $[t_i, t_{i+1}]$, we will consider each application $\mathcal{A}_k$ that can define an event in (t_i, t_{i+1}) (hence, each application $\mathcal{A}_k$ such that $t_i \geq \frac{\nu_k}{b_k}$). Then, we search for the event defined by $\mathcal{A}_k$ that maximizes the minimum yield in $[t_i, t_{i+1}]$. For that purpose, we start by looking for the best solution at time t_i. Once we have identified that solution, we determine the largest interval $[t_i, t_i'] \subset [t_i, t_{i+1}]$ such that for any $t' \in [t_i, t_i']$ the optimal solution at time t' has the same structure (which applications are allocated bandwidth, which application is allocated its maximal bandwidth, etc.) as the one at time t_i. If $t_i = t_{i+1}$ we conclude. Otherwise, we call recursively the algorithm on the interval $[t_i', t_{i+1}]$.

Because application $\mathcal{A}_k$ is defining an event at time t_i, it receives the bandwidth $\frac{\mathcal{V}_k}{t_i - t}$, where $\mathcal{V}_k$ is the remaining volume at time t (hence, the I/O operation completes at time t_i). The remaining bandwidth $B - \frac{\mathcal{V}_k}{t_i - t}$ must be distributed among the other applications. We first compute an upper-bound, y^{UB}, on the maximum minimum yield: y^{UB} is the minimum, over all applications, of the maximum yield achievable by each application at time t_i. We then check whether this upper-bound can be achieved without exceeding the total bandwidth B. In the following, let $y_j(t', b)$ denote the yield of application j at time t' if it is allocated a bandwidth of b during the interval $[t, t']$.

Assume first that y^{UB} is not achievable. Then, let $y^{opt}(t_i)$ denote the maximum minimum yield at t_i. The goal is to find the value of $y^{opt}(t_i)$ and compute the set $\mathcal{I}$ of applications to which some bandwidth must be allocated. This is exactly the set of applications whose yield ($y_j(t_i, 0)$) is strictly lower than $y^{opt}(t_i)$ if they were not allocated any bandwidth. $\mathcal{I}$ can be computed by checking the total bandwidth required for all applications to achieve a yield of at least $y_j(t_i, 0)$, for any application $\mathcal{A}_j$ in $S(t) \setminus \{k\}$. We can show that all applications receiving bandwidth must achieve the same yield. Then, knowing $\mathcal{I}$, we can compute the value of $y^{opt}(t_i)$. Hence, we know how to maximize the optimal minimum yield at time t_i (under our hypothesis, namely that $\mathcal{A}_k$ defines an event at that time). We have built a solution: $\mathcal{A}_k$ defines an event and the remaining bandwidth is distributed among the applications in $\mathcal{I}$ which all achieve the same yield. We can write the yield achieved by this solution has a function of $t' - t_i$ for $t' \in [t_i, t_{i+1}]$. This function is of the form: $a\frac{b+(t'-t_i)}{c+t'-t_i}$. Hence, it is monotonic. If it is non-increasing, we conclude that the optimal is found at time t_i. If it is increasing, we compute the latest time $t'_i \in [t_i, t_{i+1}]$ for which our solution defines the optimal solution. t'_i is the last time at which all the conditions defining the solution hold, namely, for all $t' \in [t_i, t'_i]$:

- only applications in $\mathcal{I}$ receive bandwidth: $\forall j \in S(t) \setminus \mathcal{I}, j \neq k \Rightarrow y_j(t', 0) \geq y^{opt}(t')$;
- the bandwidth limits of all applications are satisfied: $\forall j \in \mathcal{I}, \mathcal{BW}_j(t', y^{opt}(t')) \leq b_j$;
- $y^{opt}(t')$ is not greater than the yield of $\mathcal{A}_k$: $y^{opt}(t') \leq y_k\left(t', \frac{\mathcal{V}_k}{t'}\right)$.

t'_i is computed by solving a set of second degree polynomials. Then, if $t'_i = t_{i+1}$, the optimum is achieved at time t_{i+1}. Otherwise, the algorithm is called recursively on the interval $[t'_i, t_{i+1}]$.

Now, assume that y^{UB} is achievable. If the upper-bound is achieved by application $\mathcal{A}_k$, because its yield is decreasing on $[t_i, t_{i+1}]$, the optimum is achieved at time t_i. Otherwise, let $\mathcal{A}_j$ be an application whose maximum achievable yield is minimal throughout $[t_i, t_{i+1}]$ (which implies $y_j^{\max}(t_i) = y^{UB}$). Therefore, $\mathcal{A}_j$ achieves its maximum achievable yield at time t_i. Two cases can happen.

- $\mathcal{A}_j$ I/O operation ends at time t_i. Then, the optimum is achieved at time t_i because the maximum achievable yield of $\mathcal{A}_j$ is then decreasing on $[t_i, t_{i+1}]$.
- $\mathcal{A}_j$ is allocated its maximum bandwidth b_j. Then, the yield of $\mathcal{A}_j$ is increasing over $[t_i, t_{i+1}]$ (as long as we can allocate it a bandwidth of b_j). Once again, we compute the set $\mathcal{I}$ of applications to which bandwidth must be allocated. Then, we compute the latest time $t'_i \in [t_i, t_{i+1}]$ for which our solution defines the optimal solution. t'_i is the last time at which all the conditions defining the solution hold, namely:
  - only applications in $\mathcal{I}$ receive bandwidth: $\forall l \in S(t) \setminus \mathcal{I}, l \neq k \Rightarrow y_l(t', 0) \geq y_j(t', b_j)$;
  - the total bandwidth is not exceeded: $\sum_{l \in \mathcal{I}} \mathcal{BW}_l(t', y_j(t', b_j)) \leq B - \frac{\mathcal{V}_k}{t'}$;
  - the yield of $\mathcal{A}_j$ is not greater than the yield of $\mathcal{A}_k$: $y_j(y', b_j) \leq y_k\left(t', \frac{\mathcal{V}_k}{t'}\right)$.

t'_i is computed by solving a set of second degree polynomials. Then, if $t'_i = t_{i+1}$, the optimum is achieved at time t_{i+1}. Otherwise, the algorithm is called recursively on the interval $[t'_i, t_{i+1}]$.

This concludes the high-level description of BestNextEvent. All details and algorithms are available in Appendix B. Altogether, BestNextEvent is quite complicated, and admittedly too complicated for practical use. However, it will serve as a reference to help us assess the quality of the (simpler) greedy strategies of Section 7.4.1.

7.5 Lower Bounds on Competitive Ratios

This section provides lower bounds for the performance of the bandwidth-sharing strategies. The results are summarized in Table II. For instance, the first entry $m^{(1)}$ in the table means that FairShare has a competitive ratio not better than m, and that the proof of this result is given by Example 1. An entry ∞ means that the strategy does not have a ρ competitive ratio for any value of $\rho \geq 1$. Sections 7.5.1 to 7.5.6 deal with the examples that fill the lower bounds in Table II. Finally, we give some tight bounds in Section 7.5.7.

7.5.1 Example 1

We consider a window $[T_{begin}, T_{end}] = [T, T+1]$ with $T \gg 1$. The first $m-1$ applications are released at time 0, and have a yield of 1 at the beginning of the window. The m-th application is released at time T. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$, and posts an I/O operation of volume 1 at time T. FairShare allocates a bandwidth of $\frac{1}{m}$ to all applications, resulting in a yield of $\frac{T+\frac{1}{m}}{T+1}$ for the first $m-1$ applications, and a yield of $\frac{1}{m}$ for the last one. Therefore, the minimum yield is $\frac{1}{m}$.

However, if we had allocated all the bandwidth to the last application, its yield would be 1 and the minimum yield would be $\frac{T}{T+1}$. By taking T large enough, we see that FairShare has not a competitive ratio smaller than m for MinYield. This gives the first entry in Table II.

When all applications are released at time T_{begin} We now provide another example where each application $\mathcal{A}_i$ is released at the beginning of the window: $r_i = T_{begin}$ for all i. This example establishes another negative result for the competitiveness of FairShare for

	MinYield	Efficiency	Utilization
FairShare [52, 88]	$m^{(1)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$
FCFS [52, 88]	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
Set-10 [174]	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
GreedyYield	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
GreedyCom	$\infty^{(2)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$
LookAheadGreedyYield	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
PeriodicGreedyYield ($\delta \to 0$)	$2^{(4)}$	$m^{(3)}$	$\infty^{(2)}$
BestNextEvent	$\frac{m}{2} - 4^{(6)}$	$m^{(3)}$	$\infty^{(2)}$
Any strategy	$\frac{3}{2}^{(5)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$

Table II: Lower bounds for the competitive ratios of bandwidth-sharing strategies.

MinYield: there is no constant competitive ratio even when all the applications are *fresh* when entering the window.

For this example, we consider a window $[T_{begin}, T_{end}] = [0, K]$ with $K \geq 3$. The $m = K^2 + 1$ applications are all released at time 0. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$. The applications are as follows:

- $\forall i \in [1, K]$, $\mathcal{A}_i$ consists of the phases $v_i^{(0)} = \frac{1}{K+1}, W_i^{(1)} = K$;
- $\forall j \in [1, K-1], \forall i \in [Kj+1, K(j+1)]$, $\mathcal{A}_i$ consists of the phases $v_i^{(0)} = 0$, $W_i^{(1)} = j$, $v_i^{(1)} = \frac{1}{K+1}$, $W_i^{(2)} = K$;
- $\mathcal{A}_{K^2+1}$ consists of the phases $v_{K^2+1}^{(0)} = 1, W_{K^2+1}^{(1)} = K$.

Thus, the first K applications and the last one post an I/O operation at time 0, while the others start with a work phase. The schedule of FairShare is presented on the left of Figure 7.1; it is as follows:

- During the interval $[0, 1]$, applications $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K$ and $\mathcal{A}_{K^2+1}$ receive a bandwidth $\frac{1}{K+1}$. Applications $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K$ finish their I/O operation at time 1. The other applications work. Applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ finish their work at time 1.
- During the interval $[1, 2]$, applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ and $\mathcal{A}_{K^2+1}$ receive a bandwidth $\frac{1}{K+1}$. Applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ finish their I/O operation at time 2. The other applications work. Applications $\mathcal{A}_{2K+1}, \mathcal{A}_{2K+2}, \ldots, \mathcal{A}_{3K}$ finish their work at time 2.
- $\ldots$
- During the interval $[K-1, K]$, applications $\mathcal{A}_{K^2-K+1}, \mathcal{A}_{K^2-K+2}, \ldots, \mathcal{A}_{K^2}, \mathcal{A}_{K^2+1}$ receive a bandwidth $\frac{1}{K+1}$. $\mathcal{A}_{K^2-K+1}, \mathcal{A}_{K^2-K+2}, \ldots, \mathcal{A}_{K^2}$ finish their I/O operation at time K. The other applications work.

In the end, the minimum yield is the yield of application $\mathcal{A}_{K^2+1}$ which is $\frac{1}{K+1}$.

We consider the following alternative schedule (presented on the right of Figure 7.1):

- During the interval $[0, 1]$, application $\mathcal{A}_{K^2+1}$ executes its I/O operation. The other applications do nothing (even the applications that could have processed their work).
- During the interval $[1, 2]$, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K$ receive a bandwidth $\frac{1}{K+1}$. $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K$ finish their I/O operation at time 2. The other applications work. Applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ finish their work at time 2.
- During the interval $[2, 3]$, applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ receive a bandwidth $\frac{1}{K+1}$. Applications $\mathcal{A}_{K+1}, \mathcal{A}_{K+2}, \ldots, \mathcal{A}_{2K}$ finish their I/O operation at time 3. The other applications work. Applications $\mathcal{A}_{2K+1}, \mathcal{A}_{2K+2}, \ldots, \mathcal{A}_{3K}$ finish their work at

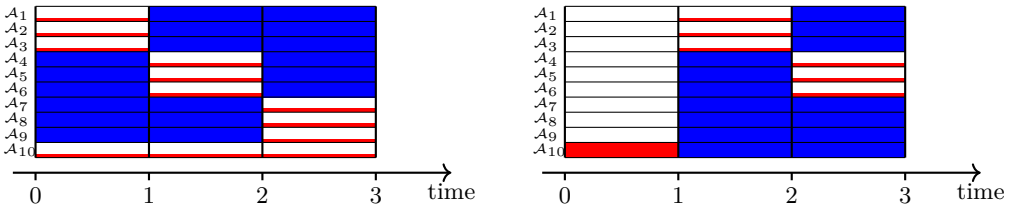


Figure 7.1: Illustration of the example for FairShare when all applications are released at time T_{begin}, with $K = 3$ (hence, $m = 10$). The schedule achieved by FairShare is depicted on the left, and the alternative schedule on the right. Blue areas represent computations, red ones I/Os, and white ones idle time. The height of a red area shows whether the I/O uses the whole bandwidth (like $\mathcal{A}_{10}$ during $[0, 1]$ in the second schedule) or which fraction of it (one fourth for all other I/Os).

time 3.

- ...
- During the interval $[K-1, K]$, applications $\mathcal{A}_{K^2-2K+1}, \ldots, \mathcal{A}_{K^2-K}$ receive a bandwidth $\frac{1}{K+1}$. $\mathcal{A}_{K^2-2K+1}, \mathcal{A}_{K^2-2K+2}, \ldots, \mathcal{A}_{K^2-K}$ finish their I/O operation at time K. The other applications work. Applications $\mathcal{A}_{K^2-K+1}, \mathcal{A}_{K^2-K+2}, \ldots, \mathcal{A}_{K^2}$ finish their work at time K.

In $[1, K]$, all applications either work or communicate. They communicate during a time at most 1, therefore they work during at least $K-2$ units of time. The minimum yield is therefore larger than $\frac{K-2}{K}$. This shows that FairShare has not a ρ competitive ratio, where $\rho \leq \frac{(K-2)(K+1)}{K}$. But $\frac{(K-2)(K+1)}{K} > K-2 = \sqrt{m-1}-2 > \sqrt{m}-3$; hence, the result that FairShare has not a $\sqrt{m}-3$ competitive ratio when all applications start execution at the beginning of the window.

7.5.2 Example 2

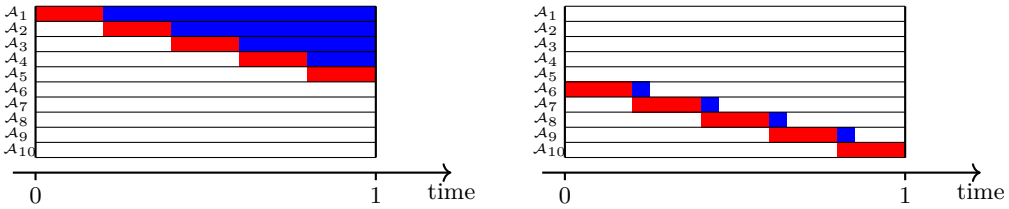


Figure 7.2: Illustration of Example 2 for the case $m = 10$ and $\epsilon = 1/5$. The best schedule for MinYield that completes the initial I/O of each application of category A (resp. of category B) is depicted on the left (resp. on the right).

We consider a window $[T_{begin}, T_{end}] = [0, 1]$. $m \geq 4$ applications are released at time 0. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$. We assume that m is even. We suppose that $m/2$ applications are in category A, i.e., have an I/O operation of volume $\frac{2}{m}$, followed by a work phase of length 1. The other $\frac{m}{2}$ applications are in category B, i.e., have an I/O operation of volume $\frac{2}{m}$ followed by a work phase of length $\alpha = \frac{\varepsilon}{m/2-1}$, where $\varepsilon > 0$ is a small number, and by another I/O operation phase of volume 1.

There are m I/O operations of volume $\frac{2}{m}$ posted at time 0. With a total bandwidth $B = 1$, it is impossible to complete more than $m/2$ of them by time 1. Because the m applications are not distinguishable at time 0, the adversary might force the scheduler to complete only I/O operations of applications of category B at time 1, and have no application of category A having completed its I/O operation by the end of the window. Proceeding with this scenario, only applications of category B may have executed some work at time 1. In fact, the most efficient scenario (which is illustrated on the right side of Figure 7.2) is to grant the full bandwidth to each application in category B one after the other, so that $m/2-1$ of them can complete their work phase by the end of the window; indeed, it is impossible for all $m/2$ applications in category B to terminate their work phase by $t = 1$, and the best, in order to maximize the work done, is to schedule the I/O operations without sharing. Finally, no application of category B can complete its second I/O transfer by $t = 1$. The efficiency at time $t = 1$ is therefore upper bounded as

$$\mathcal{E} = \frac{\sum_{i \in A \cup B} V_i^{(transferred)} + \sum_{i \in B} W_i^{(done)}}{m} \leq \frac{1}{m} + \frac{(m/2-1)\alpha}{m} \leq \frac{1+\varepsilon}{m}.$$

A strategy that would process the I/O operations of the jobs in category A without sharing

is illustrated on the left side of Figure 7.2 and would reach an efficiency

$$\mathcal{E}' = \frac{\sum_{i=1}^{m/2} \frac{i}{m/2}}{m} = \frac{m/2+1}{2m} > \frac{m}{4}\mathcal{E}$$

for ε small enough. Therefore, there is no competitive ratio lower than $\frac{m}{4}$ for EFFICIENCY for any strategy.

Now, if we consider the UTILIZATION objective function, we get $u = \frac{\epsilon}{m}$ with the first scenario and $u' = \frac{\sum_{i=1}^{m/2-1} \frac{i}{m/2}}{m} = \frac{m-2}{4m}$ with the second scenario. Therefore, we can get a competitive ratio arbitrarily large by choosing ε small enough.

Finally, for the MINYIELD objective, the best strategy is sharing I/O bandwidth equally among the $m = 2K$ applications, with gives a minimum yield of $\frac{1}{m}$. Any heuristic that serializes the I/O operations reaches a minimum yield of 0. For this example, this includes FCFS, GREEDYYIELD, GREEDYCOM, LOOKAHEADGREEDYYIELD and SET-10 (if additionally we assume that all applications belong to the same I/O-set when starting execution at time $t = 0$).

7.5.3 Example 3

We consider a window $[T_{begin}, T_{end}] = [1, 2]$. m applications are released at time 0. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$. The first $m - 1$ applications have a yield of 1 at time 1 and consists of an I/O operation of volume $\frac{\varepsilon}{m}$ followed by a work phase of length 1. The last application has an initial yield of 0 at time $t = 1$, and consist of an I/O operation of volume 1.

A schedule that executes the I/O operations of the first $m - 1$ applications in parallel completes these operations in time less than ε. These first $m - 1$ applications then work for at least $1 - \epsilon$ units of time, while the last application executes a fraction of its I/O operation. This schedule has an efficiency EFFICIENCY larger than $1 - \epsilon$.

However, GREEDYYIELD, LOOKAHEADGREEDYYIELD, PERIODICGREEDYYIELD and BESTNEXTEVENT would all allocate the whole bandwidth to the last application, because of its low initial yield. Thus they obtain an efficiency of $\frac{1}{m}$ (although they optimize the minimum yield). The same is true for FCFS, which could allocate the whole bandwidth to the last application (since all I/O operations are posted at time $t = 1$ and applications are indistinguishable). Finally, if additionally we assume all applications belong to the same I/O-set when starting execution at time $t = 1$, SET-10 might do the same and select the last application. Altogether, all these heuristics have a competitive ratio of at least m.

7.5.4 Example 4

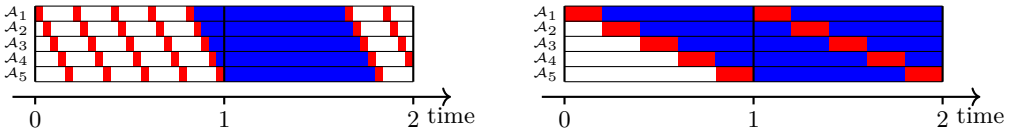


Figure 7.3: Illustration of Example 4 for the case $m = 5$ and $T = 2$. The schedule of PERIODICGREEDYYIELD ($\frac{1}{m^2}$) is depicted on the left, and an alternative schedule on the right.

We consider a window $[T_{begin}, T_{end}] = [0, T]$. m identical applications are released at time 0. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$. Each application $\mathcal{A}_i$ repeats

the same cycle indefinitely, namely an I/O operation of volume $\frac{1}{m}$ followed by a work phase of length $1 - \frac{1}{m}$.

We study PeriodicGreedyYield (δ) with $\delta = \frac{1}{m^2}$. This strategy will interleave the I/O operations of all applications in a cyclic fashion, moving from one application to the next every δ units of time. This is illustrated on the left-hand side of Figure 7.3. Without loss of generality, $\mathcal{A}_i$ will be the i-th application to complete its first I/O operation, at time $\frac{m-1}{m} + \frac{i}{m^2}$. The same cyclic sequence of I/O operations may well repeat after the first work phase, and so on until the end of the window. The minimum yield is that of $\mathcal{A}_m$, and it reaches its highest value at the end of each work phase (and then decreases during the next I/O operation). This highest value is $\frac{1}{2-\frac{1}{m}}$. Indeed, by induction, $\mathcal{A}_m$ finishes its k-th phase of work at time $k\left(1 + 1 - \frac{1}{m}\right)$, while its progress at the end of this k-th phase is equal to k.

However, a strategy that processes the I/O operations in sequence would result in a schedule with perfect yield after time 1. Such a strategy is illustrated on the right-hand side of Figure 7.3. In such a strategy, application $\mathcal{A}_i$, with $1 \leq i \leq m$ would perform I/O in the intervals $[j + \frac{i-1}{m}, j + \frac{i}{m}]$ and work in the intervals $[j + \frac{i}{m}, j + 1 + \frac{i-1}{m}]$ for all $j \geq 0$. Therefore the yield of each application will be at least $\frac{T-1}{T}$ with that strategy.

Letting T large enough, this shows that PeriodicGreedyYield (δ) has a competitive ratio at least $2 - \frac{1}{m}$ for MinYield. Letting m large enough, this shows that PeriodicGreedyYield (δ) has a competitive ratio at least 2 for MinYield.

7.5.5 Example 5

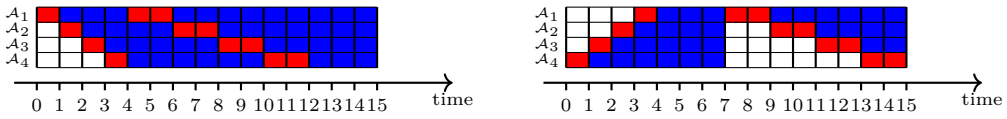


Figure 7.4: Illustration of Example 5 for the case $m = 4$. The schedule on the left maximizes MinYield. The first phase (first I/O and first computation of each application) of the schedule on the right is the best possible schedule when the first set of I/Os is completed in the worst possible order. In the second phase of the schedule, the only thing that matters is not to share the bandwidth (the order of I/O completions has no impact).

This example provides a general lower bound for the competitive ratio of any bandwidth-sharing strategy for the MinYield objective:

Lemma 30. *Given $\varepsilon > 0$, there does not exist any $\frac{3}{2} - \epsilon$ competitive algorithm for MinYield.*

Proof. We consider a set of m applications $\mathcal{A}_1, \ldots, \mathcal{A}_m$ and the window $[T_{begin}, T_{end}] = [0, 4m - 1]$. The characteristics of application $\mathcal{A}_i$, $1 \leq i \leq m$, are as follows:

- $b_i = B = 1$ and $p_i = 1$;
- $\mathcal{A}_i$ starts with an I/O operation of volume 1;
- $\mathcal{A}_i$ then has a work phase of duration $m - 2 + i$;
- $\mathcal{A}_i$ then has an I/O operation of volume 2;
- $\mathcal{A}_i$ then has a work phase of duration $3m$.

A possible schedule for each application $\mathcal{A}_i$ is the following (it is illustrated on the left-hand side of Figure 7.4):

- $\mathcal{A}_i$ waits during the time interval $[0, i - 1]$;

- $\mathcal{A}_i$ performs its initial I/O operation during the time interval $[i-1, i]$;
- $\mathcal{A}_i$ computes during the time interval $[i, m+2i-2]$;
- $\mathcal{A}_i$ performs its second I/O operation during the time interval $[m+2i-2, m+2i]$;
- $\mathcal{A}_i$ computes during the time interval $[m+2i, 4m+2i]$ (note that $4m+2i \geq 4m-1$).

All the initial I/O operations are scheduled one after the other. The same holds for all second I/O operations. Moreover, the last initial I/O operation (by $\mathcal{A}_m$) ends at time m, while the first second I/O operation (by $\mathcal{A}_1$) starts at time $m+2-2=m$. Hence, the $2m$ I/O operations are sequentialized and performed at maximal bandwidth. The application that suffers from the largest slowdown is $\mathcal{A}_m$. Its yield is $y_m = \frac{(4m-1)-(m-1)}{4m-1} = \frac{3m}{4m-1}$.

Because all m applications start with an I/O operation of volume 1, an arbitrary bandwidth-sharing strategy has no way do differentiate them. Hence, an adversary can decide that the initial I/O operations are completed in reverse order of application indices: the first completed I/O operation is that of $\mathcal{A}_m$, the second completed I/O operation is that of $\mathcal{A}_{m-1}$, and so on. Such a scenario is illustrated on the right-hand side of Figure 7.4. Then, the initial I/O operation of application $\mathcal{A}_i$ cannot finish before time $m-i+1$, because the total volume transferred when $\mathcal{A}_i$ completes its I/O operation is $m-i+1$. Then $\mathcal{A}_i$ cannot finish its work phase before time $(m-i+1)+(m-2+i) = 2m-1$. Therefore, the second I/O operation of all applications starts at or after time $2m-1$. Because the total volume of all second I/O operations is $2m$, at least one application, say $\mathcal{A}_{i_0}$, will not finish its second I/O operation before time $4m-1$, which is the end of the window. In total, this application $\mathcal{A}_{i_0}$ will have executed at most 3 units of time of I/O operation and $m-2+i_0 \leq 2m-2$ work units. Hence, the yield of $\mathcal{A}_{i_0}$ satisfies $y_{i_0} \leq \frac{2m+1}{4m-1}$.

We can then derive a bound on the competitive ratio ρ of any strategy:

$$\rho \geq \frac{\frac{3m}{4m-1}}{\frac{2m+1}{4m-1}} = \frac{3m}{2m+1} \quad \xrightarrow[m\to+\infty]{} \quad \frac{3}{2}.$$

□

7.5.6 Example 6

We consider $m > 10$ applications and a window $[T_{begin}, T_{end}] = [m, \frac{m^2}{2} - 2m + 2]$. Each application $\mathcal{A}_i$ is released at time 0, verifies $b_i = B = 1$, $p_i = 1$ and has an initial yield of 1 at time T_{begin}. The first application, $\mathcal{A}_1$, repeats the same cycle indefinitely, namely an I/O operation of volume $\frac{1}{m}$, followed by a work phase of length $\frac{\epsilon}{m^3}$. The other $m-1$ applications are identical and consist of an I/O operation of volume $1+\varepsilon$, where $\varepsilon > 0$ is a small number, followed by a work phase of length m^2.

A possible schedule would first share the bandwidth among the I/O operations of the last $m-1$ applications up to time $t = m + (m-1)(1+\varepsilon)$. The first application would then remain idle up to time t. After time t, only the first application $\mathcal{A}_1$ is posting I/O operations, so there is no further interference until the end of the window. The minimum yield would then be that of $\mathcal{A}_1$ and would verify:

$$y_1^* = \frac{T_{end} - t}{T_{end} - T_{begin}} = \frac{\frac{m^2}{2} - 2m + 2 - (m-1)(1+\varepsilon)}{\frac{m^2}{2} - 3m + 2} \geq \frac{\frac{m^2}{2} - 3m}{\frac{m^2}{2} - 2m + 2}$$

if ε is small enough (say $\varepsilon < \frac{1}{m}$).

We now study the performance of BestNextEvent. Intuitively, the parameters of the example have been chosen so that: (i) each time $\mathcal{A}_1$ posts an I/O operation, BestNextEvent assigns it the whole bandwidth immediately; and (ii) during each work

phase of $\mathcal{A}_1$, BestNextEvent assigns the whole bandwidth to the same application (the one that was granted the whole bandwidth during the first work phase of $\mathcal{A}_1$). We now prove these facts by induction on the number of events, with time $T_{begin} = m$ corresponding to event number 1.

Lemma 31. *All events taking place at a time $t < \frac{m^2}{2} - 2m$ are triggered by $\mathcal{A}_1$ (and maybe other applications). Odd-numbered events correspond to $\mathcal{A}_1$ posting an I/O operation, and even-numbered events correspond to $\mathcal{A}_1$ completing an I/O operation. Up to time t, BestNextEvent will never share the I/O bandwidth, and it will assign it as follows:*

- *$\mathcal{A}_1$ is granted the whole bandwidth at every odd-numbered event;*
- *the same application, say $\mathcal{A}_2$, is granted the whole bandwidth at every even-numbered event.*

Therefore, only $\mathcal{A}_1$ and $\mathcal{A}_2$ make some progress up to time $\frac{m^2}{2} - 2m$.

Proof. We proceed by induction on events. We assume that we are currently at the i-th event and that all previous decisions have fulfilled the conditions of the lemma so far.

Case 1: $i = 2k + 1$ for $k \geq 0$. Since the lemma is true so far, $i = 2k+1$ means application $\mathcal{A}_1$ is ready to perform an I/O operation, and $t = m + k\left(\frac{1}{m} + \frac{\epsilon}{m^3}\right) \leq \frac{m^2}{2} - 2m$, thus $k < m^3$, and the $m - 1$ last applications $\mathcal{A}_i$ for $i \geq 2$ have more than 1 unit of I/O volume remaining (by induction, because $\mathcal{A}_1$ has been working for at most k phases of length $\frac{\epsilon}{m^3}$, hence for a duration at most ε). We distinguish two cases:

Case 1.1: the next event happens in $\delta \geq 1$ units of time. In this case, we consider the yield of the application that has been assigned, in the interval $[t, t + \delta]$, the least bandwidth, b, among the last $m - 2$ applications that have not started their first I/O operation and have a current progress of m. We have $b \leq \frac{1}{m-2}$. The minimum yield at the next event would verify $y \leq f(\delta) = \frac{m+b\delta}{t+\delta}$. Using $b \leq \frac{1}{m-2}$ and $t \leq \frac{m^2}{2} - 2m$, we differentiate and find that f is non increasing. Hence, we can safely replace δ by 1 to get $y \leq \frac{m+b}{t+1} \leq \frac{m+\frac{1}{m-2}}{t+1}$.

Case 1.2: the next event happens in $\delta < 1$ units of time. In this case, the next event is defined by the completion of the I/O operation of $\mathcal{A}_1$ (the end of the window is in $T_{end} - t > 1$ time units; therefore, the next event cannot be the completion of the I/O operation of one the last $m - 1$ applications). Letting x be the fraction of bandwidth assigned to $\mathcal{A}_1$, we have $\delta = \frac{1}{mx}$. We consider the yield of the application that has the least bandwidth, b, allocated during the interval $[t, t+\delta]$, among the $m-2$ applications whose processing have not yet started (by induction): $b \leq \frac{1-x}{m-2}$. Clearly the minimum yield will not exceed $y = \frac{m+b\delta}{t+\delta} \leq \frac{m+\frac{1}{mx}\frac{1-x}{m-2}}{t+\frac{1}{mx}} = \frac{(m-2)m^2x+1-x}{(m-2)(1+mtx)}$. We let $f(x) = \frac{(m-2)m^2x+1-x}{(m-2)(1+mtx)}$. We differentiate and get $f'(x) = \frac{m^3-2m^2-mt-1}{(m-2)(1+tmx)^2}$, which is positive if m is large enough, since $t \leq \frac{m^2}{2} - 2m$. Therefore, we can safely replace x by 1 and get $y \leq \frac{m}{t+\frac{1}{m}}$. This upper bound on the minimum yield can actually be achieved by allocating all the bandwidth to $\mathcal{A}_1$; indeed, $\mathcal{A}_1$ would then have a yield of 1 and every other application a yield at least equal to $\frac{m}{t+\frac{1}{m}}$.

To conclude, we need to show that Case 1.2 results in a better minimum yield than Case 1.1, i.e., that $\frac{m+\frac{1}{m-2}}{t+1} \leq \frac{m}{t+\frac{1}{m}}$. This last inequality is equivalent to $m \geq 1 + \frac{t}{m-2} + \frac{1}{m(m-2)}$, and is true as $t \leq \frac{m^2}{2} - 2m < \frac{(m-2)^2}{2}$. As a consequence, the best decision is to allocate all the bandwidth to $\mathcal{A}_1$, which concludes the induction step for $i = 2k + 1$.

Case 2: $i = 2k$ **for** $k > 0$**.** Since the lemma is true so far, the event $i = 2k$ occurs at time $t = m + (k-1)\left(\frac{1}{m} + \frac{\epsilon}{m^3}\right) + \frac{1}{m} \leq \frac{m^2}{2} - 2m$. Hence, again, $k < m^3$ and $\mathcal{A}_1$ is starting a work phase. If $k = 1$ we will show that the whole bandwidth is assigned to one application, say $\mathcal{A}_2$. If $k > 1$, by induction $\mathcal{A}_2$ has transferred a volume $\frac{(k-1)\epsilon}{m^3} < \epsilon$ so far. Regardless of the value of k, by induction, any application $\mathcal{A}_i$ with $i \geq 3$ has not started its execution.

Case 2.1: The next event is the end of the window. In this case, let b be the minimum bandwidth allocated to an application other than $\mathcal{A}_2$ (and $\mathcal{A}_1$) during the interval $[t, t+\delta]$. Thus, $b \leq \frac{1}{m-2}$ and $\delta \geq T_{end} - t > 2$ does not depend on b. We get $y \leq f(\delta) = \frac{m + \frac{1}{m-2}\delta}{t+\delta}$. We obtain $f'(\delta) = \frac{-m^2+2m+t}{(m-2)(\delta+t)^2} < 0$ (because $t < \frac{m^2}{2} - 2m$). We replace δ by 2 and get $y \leq \frac{m + \frac{2}{m-2}}{t+2}$.

Case 2.2: The next event is not the end of the window. If the next event is not the end of the window, let $\mathcal{A}_{i_0}$ be the application defining the next event at time $t + \delta$ and let x be the bandwidth allocated to $\mathcal{A}_{i_0}$ (we will eventually show that $\mathcal{A}_{i_0} = \mathcal{A}_2$). Let $\mathcal{A}_{i_1}$ be the application with smallest bandwidth, b, allocated to it during $t, t+\delta]$, among the applications that are not $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_{i_0}$. Therefore, $b \leq \frac{1-x}{m-3}$ and by assumption $\mathcal{A}_{i_1}$ has not started. The minimum yield will not exceed the yield of $\mathcal{A}_{i_1}$, which is $y = \frac{m+b\delta}{t+\delta}$. We distinguish two last sub-cases:

Case 2.2.1: $i_0 = 2$**.** We have $\delta = \frac{1+\epsilon-\frac{(k-1)\epsilon}{m}}{x}$, and $y \leq y(\mathcal{A}_{i_1}) \leq f(x) = \frac{m + \frac{1-x}{m-3}\frac{1+\epsilon-\frac{(k-1)\epsilon}{m}}{x}}{t + \frac{1+\epsilon-\frac{(k-1)\epsilon}{m}}{x}}$.

We obtain $f'(x) = \frac{(\epsilon(-k+m+1)+m)\left(\epsilon(k-m-1)+m\left(m^2-3m-t-1\right)\right)}{(m-3)(\epsilon(-k+m+1)+mtx+m)^2}$. If ϵ is small enough $f'(x)$ will have the sign of $m^2(m^2-3m-t-1)$ which is positive as $t \leq \frac{m^2}{2} - 2m$. We can safely bound $f(x)$ by $f(1)$ and the minimum yield will verify $y \leq \frac{m}{t+1+\epsilon-\frac{(k-1)\epsilon}{m}}$. We point out that this bound on the minimum yield is achievable by allocating all the bandwidth to application $\mathcal{A}_2$; hence, we have an equality.

Case 2.2.2: $i_0 \neq 2$**.** In this case, we have $\delta = \frac{1+\epsilon}{x}$, and $y \leq y(\mathcal{A}_{i_1}) \leq f(x) = \frac{m + \frac{1-x}{m-3}\frac{1+\epsilon}{x}}{t + \frac{1+\epsilon}{x}}$. We obtain $f'(x) = \frac{(\epsilon+1)\left(m^2-3m-t-1-\epsilon\right)}{(m-3)(\epsilon+tx+1)^2}$. If ϵ is small enough, $f'(x)$ will have the sign of $m^2 - 3m - t - 1$, which is strictly positive as $t \leq \frac{m^2}{2} - 2m$. We can safely bound $f(x)$ by $f(1)$ and the minimum yield will verify $y \leq \frac{m}{t+1+\epsilon}$. Note that, once again, this bound on the minimum yield is achievable by allocating all bandwidth to application $\mathcal{A}_{i_0}$; hence we have an equality.

A quick computation shows that if $k = 1$, cases 2.2.1 and 2.2.2 are equivalent and better than 2.1 because $\frac{m}{t+1+\epsilon} > \frac{m+\frac{2}{m-2}}{t+2} \Leftrightarrow m(m-2) > \frac{2(t+1+\epsilon)}{1-\epsilon}$, which is true if ϵ is small enough. Therefore, if $k = 1$, the algorithm allocates all the bandwidth to an arbitrarily chosen application. When $k > 1$, Case 2.2.1 achieves a strictly better yield than the other two cases. Hence, BestNextEvent will always chose to allocate bandwidth to the same application.

This concludes the proof of the lemma.

□

A direct consequence of Lemma 31 is that each application $\mathcal{A}_i$ with $i \geq 3$ will not

have any progress in the interval $[m, \frac{m^2}{2} - 2m]$. Therefore, the minimum yield is at most $y' = \frac{m+2}{\frac{m^2}{2}-2m+2} = \frac{2(m+2)}{m^2-4m+4}$. Therefore, $\frac{y^*}{y'} \geq \frac{m^2-6m}{2(m+2)} > \frac{m}{2} - 4$.

7.5.7 Tight Bounds

This section provides additional results on the performance of the bandwidth-sharing strategies.

The competitive ratio of FairShare is exactly m for MinYield

Section 7.5.1 has shown that the competitive ratio of FairShare for the MinYield objective is at least m. In fact, this competitive ratio is exactly m. Intuitively, with FairShare, each application progresses at full rate when computing, and at least at the fraction $\frac{1}{m}$ of the optimal rate when performing an I/O operation.

To see this formally, whenever an application $\mathcal{A}_i$ posts an I/O operation at time t, it receives either its maximal bandwidth b_i or the fair fraction $\frac{b_i}{\sum_{j \in S(t)} b_j} \geq \frac{b_i}{m}$. Therefore, if $\mathcal{A}_i$ was released at time r_i with an initial yield $y_i(T_{begin})$ at the beginning of the window $[T_{begin}, T_{end}]$, we get

$$\begin{aligned} y_i(T_{end}) &\geq \frac{y_i(T_{begin}) \times (T_{begin} - r_i) + \frac{T_{end} - T_{begin}}{m}}{T_{end} - r_i} \\ &\geq \frac{y_i(T_{begin}) \times (T_{begin} - r_i) + (T_{end} - T_{begin})}{m(T_{end} - r_i)} \geq \frac{y_i^{opt}}{m}, \end{aligned}$$

where y_i^{opt} is the best achievable yield for $\mathcal{A}_i$.

The competitive ratios of FCFS, Set-10, GreedyYield, LookAheadGreedyYield, and PeriodicGreedyYield are exactly m for Efficiency

Any bandwidth-sharing strategy that always allocates the whole bandwidth that can be allocated (regardless of the details of the allocation) does achieve a competitive ratio of m for Efficiency. Indeed, if there is no I/O operation at time t, the efficiency is 1, and if there is some I/O operation, the efficiency is at least $\frac{1}{m}$. The five strategies listed here do allocate the whole possible bandwidth at each event. The lower bounds come from Section 7.5.3.

FairShare can be arbitrarily better than BestNextEvent

In the example of Section 7.5.6, the I/O operations of applications $\mathcal{A}_i$ for $i > 1$ will always receive a fraction of the total bandwidth greater than $\frac{1}{m}$ with FairShare. These applications will complete their I/O operations at time at most $m + m(1 + \epsilon) < 3m$. After this point, there would not be further interference, and the minimum yield y^* for FairShare verifies

$$y^* \geq \frac{\frac{m^2}{2} - 5m}{\frac{m^2}{2} - 2m + 2} > \left(\frac{m}{2} - 6\right) \frac{2(m+2)}{(m-2)^2} \geq \left(\frac{m}{2} - 6\right) y_N,$$

where y_N is the minimum yield achieved by BestNextEvent, whose upper bound $\frac{2(m+2)}{(m-2)^2}$ is given in Section 7.5.6.

7.6 Performance Evaluation

We first formally define the main parameter for the experiments, namely the I/O pressure, in Section 7.6.1. Then, we detail the simulations conducted with synthetic traces in Section 7.6.2 before discussing results for the APEX workloads in Section 7.6.3. This code is publicly available at `https://gitlab.inria.fr/luperoti/BandwithStrategies`.

7.6.1 I/O Pressure

For a given a steady-state window $[T_{begin}, T_{end}]$ with m applications, we compute the volume V_i that each application $\mathcal{A}_i$ would be able to transfer if it was executed in dedicated mode throughout the window. The total I/O volume to transfer during the window is $V = \sum_{i=1}^{m} V_i$. The I/O pressure W is then

$$W = \frac{V}{B(T_{end} - T_{begin})}. \tag{7.11}$$

The I/O pressure W is the ratio of this total volume V over the maximum volume that could have been transferred during the window, assuming that it consists of a single block of data available at T_{begin}. Of course, if W exceeds 1, some transfers will necessarily be delayed. But even if W is lower than 1 but high, say 0.8, it is likely that I/O interferences and delays due to work phases will prevent to transfer the whole data volume V before the end of the steady-state window.

The I/O pressure W is a key parameter for the simulations: most bandwidth-strategies are expected to perform well when W is low, but we aim to assess how much their performance drops when W is high.

7.6.2 Synthetic Traces

Framework

The synthetic traces follow the methodology of [30] and consist of $m = 60$ applications, each of them being able to saturate the bandwidth (we have $b_i = B = 1$), with an approximate horizon of $h = 2,000,000$. For a given aimed pressure W^{GOAL}, each application $\mathcal{A}_i$ $(1 \leq i \leq m)$ is defined by the three parameters $(\mu_i, \sigma'_i, \nu_i)$: μ_i and σ'_i represents expectation and standard deviation and impact the length of the repetitions for each applications and ν_i determines how much the application differs from one iteration to another. More precisely,

- We generate an iteration duration ω_i for $\mathcal{A}_i$, which corresponds to the sum of a work phase and an I/O phase if the application was alone on the platform. This duration is generated using the two parameters μ_i and σ'_i: ω_i is drawn from the normal distribution $\mathcal{N}(\mu_i, \sigma'_i)$, truncated so that we consider only positive results.
- The number of iterations of application $\mathcal{A}_i$ is $n_i = \left\lceil \frac{h}{\omega_i} \right\rceil$ so its total completion time if it were alone on the platform is close to h.
- All applications are released at time T_{begin}: $r_i = T_{begin}$ for each application $\mathcal{A}_i$. In other words, all applications are *fresh* when entering the window and have the same yiled (equal to 0). To avoid having all applications synchronized, we add a work phase $w_i^{(0)}$ whose length is generated in $\mathcal{U}[0, \omega_i]$, so that application $\mathcal{A}_i$ effectively starts at time $w_i^{(0)}$. To simulate Set-10, we put all applications in the same I/O-set with highest priority initially, and hence process them in FCFS order at the beginning of

the execution. After that each application has completed its first I/O operation, the duration of each iteration is updated on the fly, and applications get classified into different I/O-sets.

- Next, for each application, we fix the time spent on I/Os vs. on computing, so that the total pressure is around W^{GOAL}. This is done by drawing a value u_k uniformly at random in $\mathcal{U}[0,1]$ for each application $\mathcal{A}_k$ ($1 \leq k \leq m$), and then by defining the fraction of I/O for application $\mathcal{A}_i$ as $\phi_i = \frac{u_i W^{GOAL}}{\sum_{k=1}^{m} u_k}$. This guarantees that the I/O pressure W is around W^{GOAL}. Indeed, ϕ_i allows us to define the average duration of computing phases $t_{i,cpu} = (1-\phi_i)\omega_i$ and the average volume of I/O phases: $t_{i,io} = \phi_i \omega_i$. Thus

$$\begin{aligned} W &\approx \frac{\sum_{i=1}^{m} t_{i,io} n_i}{B(T_{end} - T_{begin})} = \frac{\sum_{i=1}^{m} \phi_i \omega_i n_i}{B(T_{end} - T_{begin})} \\ &\approx \frac{\sum_{i=1}^{m} \phi_i T_{end}}{B(T_{end} - T_{begin})} = \frac{T_{end} W^{GOAL}}{T_{end}} = W^{GOAL}. \end{aligned}$$

  We point out that we cannot enforce exactly $W = W^{GOAL}$ due to the randomness in the generation of instances.
- Finally, for each application $\mathcal{A}_i$, we consider a noise parameter ν_i to generate iterations of different lengths. For all $j \leq n_i$, we draw two variables $\gamma_{cpu}^{(j)}$ and $\gamma_{io}^{(j)}$ from a uniform distribution $\mathcal{U}[-\nu_i, \nu_i]$ and let $W_i^{(j)} = (1+\gamma_{cpu}^{(j)}) t_{i,cpu}$ and $v_i^{(j)} = (1+\gamma_{io}^{(j)}) t_{i,io}$.

Results for Synthetic Traces

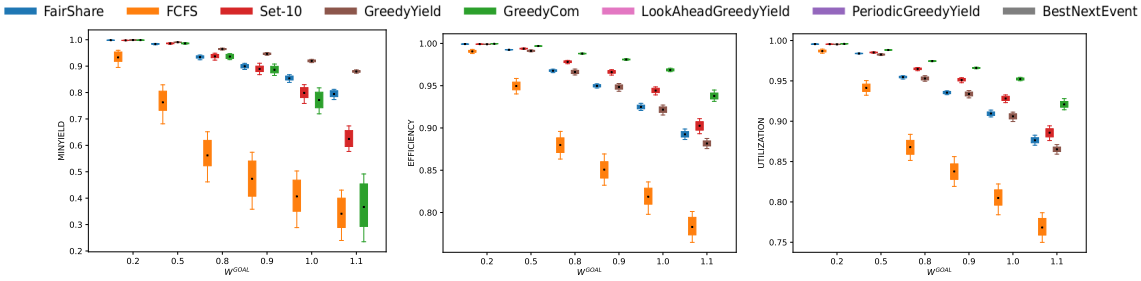
Still following the methodology of [30], the experiments are conducted by varying four different key parameters for the 60 applications. For the application length, we consider 20 applications of medium size, and then a proportion of smaller and larger applications, as determined by the parameter n_{small} (number of small applications). The standard deviation is dictated by parameter σ, the noise is set to ν, and the pressure is W^{GOAL}. Overall, the applications are as follows:

- n_{small} *small* applications with parameters $(\mu = 1\,000, \sigma' = \mu\sigma, \nu)$;
- 20 *medium* applications with parameters $(\mu = 10\,000, \sigma' = \mu\sigma, \nu)$;
- $40 - n_{small}$ *big* applications with parameters $(\mu = 100\,000, \sigma' = \mu\sigma, \nu)$.

The time window is defined as $[T_{begin} = 0, T_{end} \approx h]$, where T_{end} is the smallest time required to complete an application when it is running alone on the platform. Each application is generated in such a way that T_{end} is approximately equal to $h = 2,000,000$. For each set of experiments, we study the results of all the heuristics for the three objectives (MinYield, Efficiency, Utilization).

Finally, for each set of parameters, we generate $K = 200$ instances on which we test all the heuristics presented in Section 7.4 (including the reference heuristics FairShare, FCFS and Set-10). In the following sections, we vary the parameters one by one and present the results on different figures. Each set of instances is represented by a boxplot of a color associated with the studied heuristic. In these boxplots, the 25th and 75th percentiles of the K instances delimit the box, and the 10th and 90th percentiles are at the end of the whiskers. Finally, the boxplots are connected by a line passing through their means.

Impact of the target I/O pressure (W^{GOAL}). We first set $\nu = \sigma = 0.5$, and $n_{small} = 20$ (20 applications of each category), and we present the results of the experiments for all



(a) MinYield (b) Efficiency (c) Utilization

Figure 7.5: Impact of the aimed I/O pressure (W^{GOAL}).

values of the aimed I/O pressure $W^{GOAL} \in [0.2, 0.5, 0.8, 0.9, 1.0, 1.1]$ on Figure 7.5. As soon as the pressure increases, we see that the state-of-the-art strategies FairShare, FCFS, and Set-10, along with the new GreedyCom, fail to keep a minimum yield close to 1. The other newly proposed strategies, which all focus on the yield, successfully maintain a very high minimum yield, and achieve a similar performance which very slowly degrades when the I/O pressure increases. LookAheadGreedyYield and BestNextEvent achieve a very slightly worse performance than GreedyYield and PeriodicGreedyYield. This counter-intuitive result may be explained by the fact that an I/O phase is always followed by a computation phase during which the progress rate of an application is perfect. Hence, what heuristics GreedyYield and PeriodicGreedyYield may loose in terms of application yield during an I/O phase may be made up later on in the subsequent computation phase. The fact that GreedyYield, PeriodicGreedyYield and LookAheadGreedyYield achieve a minimum yield no worse than that of BestNextEvent, a costly strategy which exhaustively looks for the best solution, strongly validates these three low-cost strategies.

The classical FCFS strategy also has very poor results in terms of efficiency and utilization, while GreedyCom is actually the best for these objective functions since it will complete short I/Os first, with a risk of starvation for applications with long I/Os. This explains the poor performance of GreedyCom for MinYield for higher values of W^{GOAL}. The yield-based strategies tend to balance the yield of all applications, which optimizes the MinYield. However, not allowing any application to starve requires prioritizing some long I/Os that saturate the bandwidth, which can negatively impact both Efficiency and Utilization. The underlying tradeoff explains why heuristics achieving a significantly better performance than FairShare for the MinYield usually achieve slightly worse performance than FairShare for Efficiency and Utilization. However, the performance degradation in terms of either Efficiency or Utilization is quite small (under 5%) and only happens for the largest value of I/O pressure. For all but the largest value of W^{GOAL}, LookAheadGreedyYield even achieves better Efficiency and Utilization than FairShare.

Impact of iteration size (ω) and I/O fraction (ϕ). We investigate the impact of ω and ϕ on the yield of each individual application for a fixed set of parameters: $\nu = 0.5$, $\sigma = 0.5$, $n_{small} = 20$, and $W^{GOAL} \in \{0.5, 0.8, 1.1\}$. More precisely, for each experiment $\mathcal{E}_k$, we define two permutations on the index set $\{1, 2, \ldots, 60\}$ to sort the applications by increasing values of ω (permutation π^k_ω) or of ϕ (permutation π^k_ϕ). For each value of W^{GOAL}, we compute the average yield of applications in each position i under each

permutation, denoted as $\overline{y}_i^{(\omega)}$ (resp. $\overline{y}_i^{(\phi)}$), for $i \in \{1, 2, \ldots, 60\}$. It is computed as follows:

$$\overline{y}_i^{(\omega)} = \frac{1}{K} \sum_{k=1}^{K} y_{\pi_\omega^k(i)} \quad \text{and} \quad \overline{y}_i^{(\phi)} = \frac{1}{K} \sum_{k=1}^{K} y_{\pi_\phi^k(i)}.$$

We then plot the value of $\overline{y}_i^{(\omega)}$ for i varying in $[1, 60]$ on Figure 7.6 and the value of $\overline{y}_i^{(\phi)}$ on Figure 7.7. Therefore, the leftmost point on Figure 7.6 (respectively, on Figure 7.7) corresponds to the average yield of the application with the smallest value of ω (resp., of ϕ), while the rightmost point corresponds to the average yield of the application with the largest value of ω (resp., of ϕ).

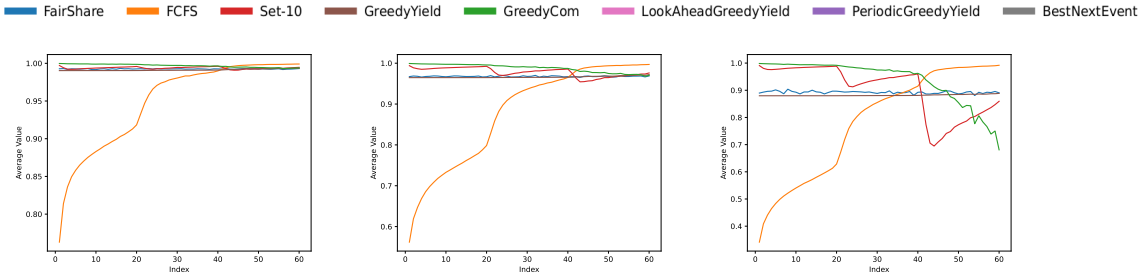
Impact of iteration size (ω). On Figure 7.6, we observe that the differences between the heuristics are more pronounced when W^{GOAL} increases. This is because the increase in W^{GOAL} increases the I/O interferences. For this reason, we now focus on the figure on the right (case $W^{GOAL} = 1.1$). First, we can observe that the variation of ω has little impact on the yields achieved by FairShare, GreedyYield, LookAheadGreedyYield, PeriodicGreedyYield, and BestNextEvent. GreedyYield, LookAheadGreedyYield PeriodicGreedyYield, and BestNextEvent tend to balance the yield of the different applications, resulting in a constant function. For FairShare, there seems to be no correlation between ω and the yield. This can be explained by the fact that there is no correlation between ω and ϕ in the generated instances.

This figure is more enlightening for the other heuristics. First, the yield seems to be positively correlated with ω for FCFS. This is because ϕ is not correlated with ω. Hence, a small value of ω corresponds to short I/O phases. For FCFS, the longest I/Os will saturate the bandwidth more often. Indeed, a single application can saturate the bandwidth, so when a long I/O is executed, all the other applications wanting to performa some I/O are stopped. For an application with a small I/O, the waiting time may be very long compared to its size, and the next waiting phase may also come quickly if some long I/O is posted between two of its I/O phases. Therefore, applications with short I/Os, i.e., a small value of ω, will spend a large part of their time waiting.

We observe the opposite behavior for the GreedyCom strategy since, this time, small I/Os are given priority. As previously mentioned, a low value of ω induces short I/Os; hence, the yield decreases with ω.

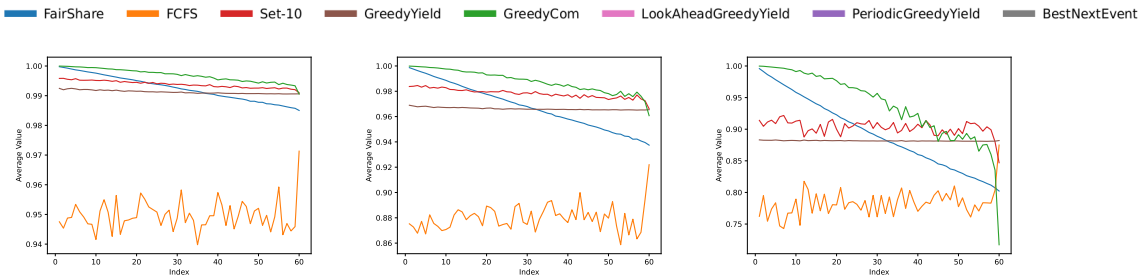
Finally, this figure perfectly illustrates the behavior of Set-10. Indeed, we can clearly distinguish the three steps corresponding to the three priority categories in these synthetic traces. Moreover, within each of these steps, we see that the yield increases with ω, just like for FCFS. This is because Set-10 behaves like FCFS inside each of these categories.

Impact of I/O fraction (ϕ). Figure 7.7 may appear a bit more cluttered, but illustrates some interesting behaviors. Once again, we only focus on the figure on the right, that is, on the case $W^{GOAL} = 1.1$. First, we can see a difference between GreedyYield (hidden under PeriodicGreedyYield) and LookAheadGreedyYield for small values of ϕ, showing that the best immediate choice is not always the best choice in the long term. We can also see that BestNextEvent favors applications with smaller I/Os so that the next event arrives as soon as possible and the yield do not have the time to significantly decrease (because of I/O interference). FCFS is erratic because an application with a large ω but a small ϕ will still have larger I/O volumes per phase than an application with a small ω but a large ϕ. The same argument also explains the non-monotonic behavior of GreedyCom when ϕ becomes large. The only heuristic that is strongly (negatively) correlated with ϕ is



(a) $W^{GOAL} = 0.5$ (b) $W^{GOAL} = 0.8$ (c) $W^{GOAL} = 1.1$

Figure 7.6: Yields sorted by iteration size (ω).

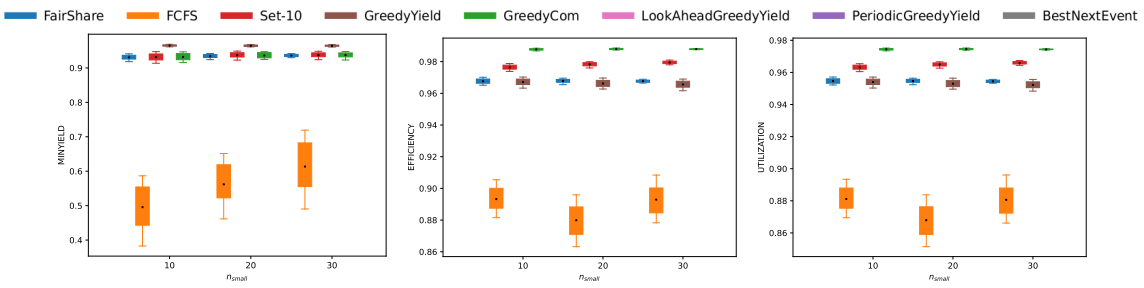


(a) $W^{GOAL} = 0.5$ (b) $W^{GOAL} = 0.8$ (c) $W^{GOAL} = 1.1$

Figure 7.7: Yields sorted by I/O fraction (ϕ).

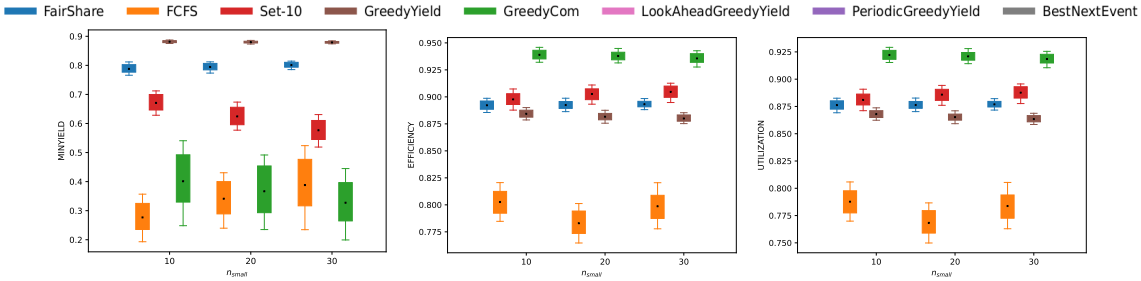
FairShare. Indeed, the larger ϕ, the longer the application will spend performing I/Os, and the lower the yield will be, whereas in a working phase, the instantaneous yield is 1. The linear shape of this curve is related to the uniform distribution of ϕ.

Impact of the number of small applications n_{small}. We let $\nu = \sigma = 0.5$, and run experiments for all values of $n_{small} \in [10, 20, 30]$, both in the low I/O pressure scenario (Figure 7.8, with $W^{GOAL} = 0.8$) and in the high I/O pressure scenario (Figure 7.9, with $W^{GOAL} = 1.1$). The results are pretty similar for all values of n_{small}, and we draw the same conclusions as in Section 7.6.2), in particular when the I/O pressure is high. Indeed, with a low I/O pressure, almost all heuristics succeed to achieve a high value of MinYield.



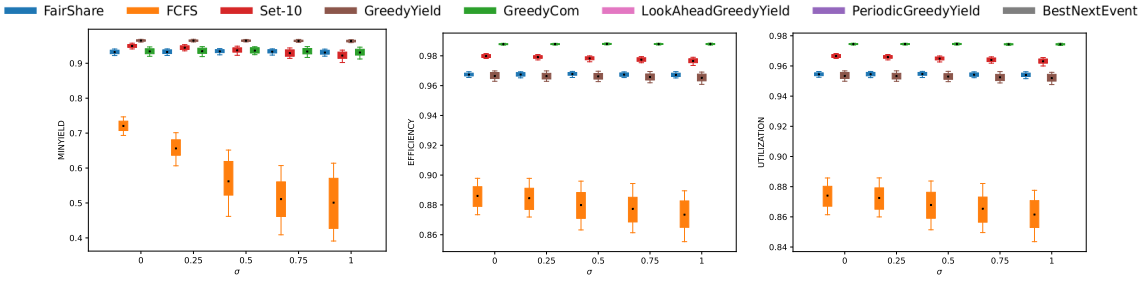
(a) MinYield (b) Efficiency (c) Utilization

Figure 7.8: Impact of the number of small tasks (n_{small}) with low I/O pressure ($W^{GOAL} = 0.8$).



(a) MinYield (b) Efficiency (c) Utilization

Figure 7.9: Impact of the number of small tasks (n_{small}) with high I/O pressure ($W^{GOAL} = 1.1$).

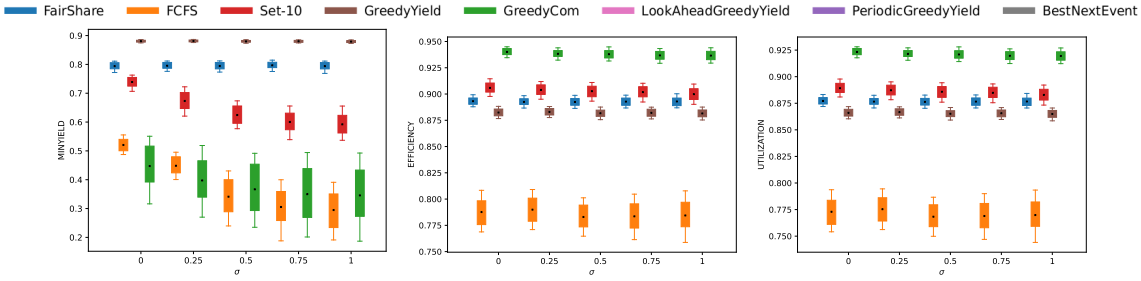


(a) MinYield (b) Efficiency (c) Utilization

Figure 7.10: Impact of the standard deviation (σ) with low I/O pressure ($W^{GOAL} = 0.8$).

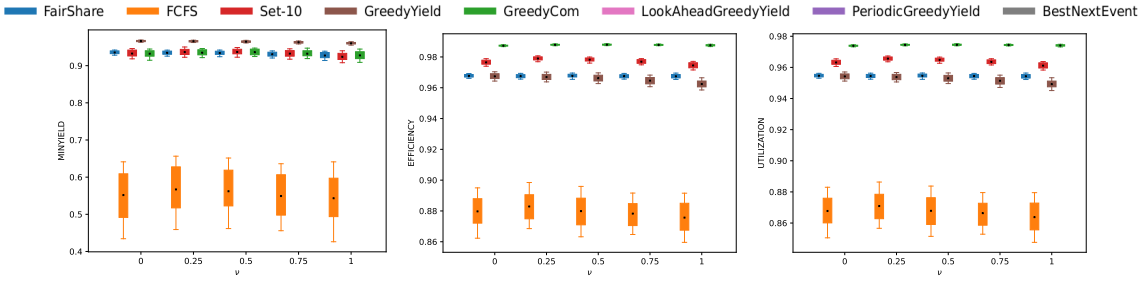
Impact of the standard deviation σ. We set $\nu = 0.5$, $n_{small} = 20$, and consider $\sigma \in [0, 0.25, 0.5, 0.75, 1]$, both in the low I/O pressure scenario (Figure 7.10, with $W^{GOAL} = 0.8$) and in the high I/O pressure scenario (Figure 7.11, with $W^{GOAL} = 1.1$). We observe that the novel heuristics (except maybe GreedyCom when the I/O pressure is high) are not affected by an increase in the standard deviation σ, while FCFS suffers from high standard deviation. In the scenario with a high I/O pressure, the MinYield achieved by Set-10 significantly decreases when σ increases.

Impact of the noise. In these experiments, we set $\sigma = 0.5$, $n_{small} = 20$, and we consider values $\nu \in [0, 0.25, 0.5, 0.75, 1]$, both in the low I/O pressure scenario (Figure 7.12, with $W^{GOAL} = 0.8$) and in the high I/O pressure scenario (Figure 7.13, with $W^{GOAL} = 1.1$). We see that the noise does not affect any of the heuristics, which are resilient to variations



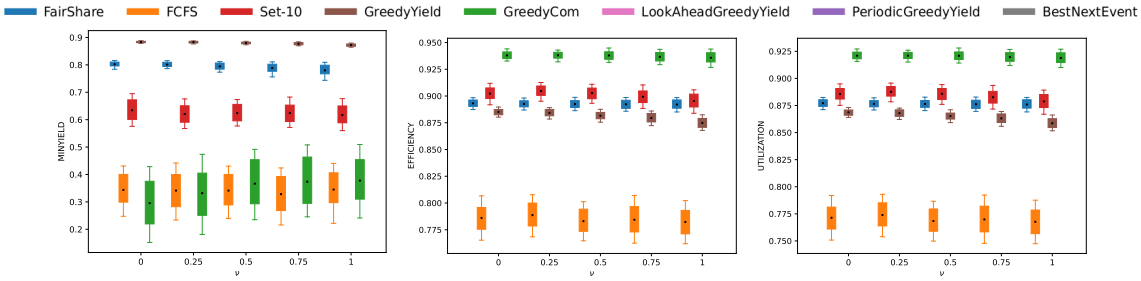
(a) MinYield (b) Efficiency (c) Utilization

Figure 7.11: Impact of the standard deviation (σ) with high I/O pressure ($W^{GOAL} = 1.1$).



(a) MinYield (b) Efficiency (c) Utilization

Figure 7.12: Impact of the noise (ν) with low I/O pressure ($W^{GOAL} = 0.8$).



(a) MinYield (b) Efficiency (c) Utilization

Figure 7.13: Impact of the noise (ν) with high I/O pressure ($W^{GOAL} = 1.1$).

in the lengths of the working and I/O phases.

Synthesis of the Evaluation on Synthetic Scenarios

For the MinYield objective, the greedy strategies LookAheadGreedyYield, GreedyYield and PeriodicGreedyYield achieve comparable performance, and much better performance than the competitors FCFS, FairShare and Set-10. Furthermore, we stress that the complicated strategy BestNextEvent does *not* turn out to be superior to the simpler ones, which is good news: GreedyYield, LookAheadGreedyYield and PeriodicGreedyYield are all simple to implement and use. Finally, for the Efficiency and Utilization objectives, GreedyCom is the best, FCFS is the worst, and the other strategies achieve close performance in between.

7.6.3 Evaluation on APEX workloads

Apex Traces [109]

We use the workload and platform described in [109] to evaluate the bandwidth-sharing strategies on realistic scenarios. The table in Figure 3 of [109] describes two very different workloads: the NERSC workload and the TRILAB workload. The NERSC workload contains a large number of small applications (e.g., a single pipeline of the SkySurvey workflow runs over 24 cores for 4 hours, but the set of SkySurvey workflows represents 12% of the overall core-hours used by the workload on the machine), some large applications (GTS spans over 16,512 cores, or 1/8 of the platform, for 48 hours), and some very long running applications (CESM applications run for 10 days over 8,000 cores). The TRILAB workload contains a more homogeneous set of applications (4096 to 32768 cores), and

all applications run for a significantly longer time (64 hours for the smallest duration, and up to 12 days for the longest). From this table, we take the application walltime, its number of cores, and the data information to build a possible schedule on the target machine. The table reports how much input, output, and checkpoint data each application uses. The trace does not provide fine-grain information on how the data is consumed or produced. To simulate the schedules, we assume that all inputs happen at the beginning of the application, which then does periodic checkpoints, and eventually outputs all its output data just before its completion. As is often the practice in HPC centers [80], we use a fixed period of 1 hour for the checkpoint interval.

Based on this information, we generate machine schedules using the first-fit strategy. We consider independently NERSC or TRILAB workloads and, for a given workload, we randomly pick applications from this workload, and place them on the schedule, until two conditions are met: 1) the schedule follows the application workload distribution described in the APEX table, and 2) the schedule represents at least 3 months of machine use. For each target machine considered (see below), we generate 100 schedules for the TRILAB workload and 100 schedules for the NERSC workload. In each schedule, we then find the 20 longest windows during which no application is joining or leaving the machine, to fit the analysis conditions with steady-state windows described in Sections 7.3 and 7.4. We then assume that each application joined the system at the window start ($r_i = T_{begin}$ for each application $\mathcal{A}_i$).

On the Celio system[2], both the NERSC and TRILAB workloads represent a small I/O pressure (about 0.15 on average). However, I/O pressure is a metric that tends to increase as we consider larger platforms and newer systems. In [104], the authors look at the architectural trends and system balance of the top 500 supercomputers. The Parallel File System (PFS) bandwidth is studied for systems that existed between 2009 and 2018. The authors compare the PFS bandwidth with the aggregated memory bandwidth. The different systems have a ratio of aggregated memory bandwidth by PFS bandwidth between 50 and 17000, with an average of 13,353, without a clear trend in time.

The ratio of aggregated memory bandwidth per computing performance, however, shows a clear diminishing trend. As an example, this ratio decreased by a factor 9 between the No. 1 machine in 2009 and the No. 1 machine in 2018. As a consequence, the ratio between the PFS bandwidth and the computing performance also has a clearly decreasing trend. In [30], the authors note that this ratio has decreased by a factor 24.8 over 20 years. Over long periods of time, it looks like the trend of the PFS bandwidth progresses more slowy than the computing power by a linear factor.

To study how the different algorithms behave with higher values of the I/O pressure, we have considered a set of target machines that are scaled versions of the Celio system. Let C_c and C_{bw} be respectively the total number of cores and system bandwidth of Celio, and let t represent the passing time. The system M_t has $C_c \times 2^{\frac{t}{\alpha}}$ cores (representing a doubling of computing power every α time units, in accordance of the observed progression in [104]), and $C_{bw} \times 2^{\frac{t}{\alpha}}/t$ system bandwidth, following the observation above. M_y, $y > 0$, represents machines built y time units later than the Celio machine, and for each target machine, we compute the schedules and corresponding windows for both workloads. We thus obtain a range of I/O pressures between 0.15 and 1.4, and simulate the behavior of the bandwidth-sharing strategies in each window, to evaluate our metrics as a function of the I/O pressure.

[2]Celio is the platform used for the NERSC and TRILAB workloads [109].

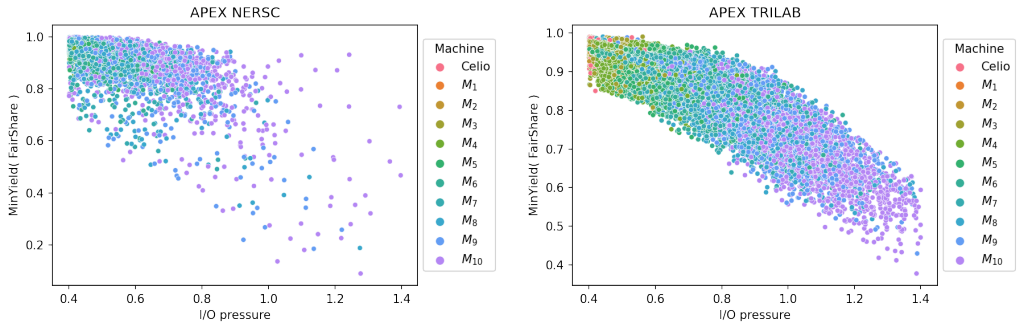


Figure 7.14: MinYield of the FairShare strategy for the NERSC and TRILAB workloads, as a function of the I/O pressure and of the target platform.

MinYield of FairShare on APEX Scenarios

We use the FairShare strategy as the basis for our evaluation, so we study first how FairShare behaves as a function of the I/O pressure. Figure 7.14 presents the MinYield obtained by the FairShare strategy within each of the 2,000 windows obtained during the simulation, as a function of the I/O pressure observed inside each window. The color of points denote on which target platform this I/O pressure and MinYield were observed.

On the NERSC workload, we see that the MinYield stays above 0.8 when the I/O pressure is low (0.4), and the distribution tends to decrease as the I/O pressure increases, with some scenarios that obtain a MinYield under 0.5 when the I/O pressure is 1, and the number of runs that have a low MinYield continue to increase as the I/O pressure continues to increase. The machine scale has some impact on the I/O pressure inside the various windows, but most of the runs present a relatively low I/O pressure, and a MinYield of 1 for FairShare is observed for some runs with high I/O pressures (up to 1.4). We conjecture that this is a consequence of the relatively small windows for the NERSC workload. Small scale, short lived applications constitute the bulk of many windows of the NERSC workload. These applications only do I/O at the beginning and end of their execution, limiting the opportunities for interferences. These I/O are also small (even relative to the short duration of the application), so when they interfere (which is unavoidable when the I/O pressure is higher than 1), they still reduce the MinYield by only a fraction. Only on windows that feature the few larger applications and those with costly checkpoints can we observe a measurable decrease of MinYield for FairShare.

This conjecture is corroborated by the measurement of the TRILAB workload. The same trends for this workload are more clearly marked: the larger the machine, the higher the I/O pressure, and the higher the I/O pressure, the lower MinYield for FairShare. Although there are no scenarios where MinYield goes under 0.4, there are also no scenarios with a MinYield close to 1 when the I/O pressure is above 1. The windows are much longer in the TRILAB experiments, and applications have time to checkpoint regularly during these windows. As a consequence, interferences between applications that have overlapping I/O create slowdowns that reduce the MinYield. We note from the left graph of Figure 7.14 that no NERSC scenario on the Celio platform obtains an I/O pressure of at least 0.5, while some scenarios of the TRILAB workload can saturate the I/O bandwidth.

MinYield of All Strategies on APEX NERSC Scenarios

Figure 7.15 presents all the scenarios used in Figure 7.14 for the NERSC workload, and considers the MinYield of each strategy as a ratio of MinYield for FairShare, with

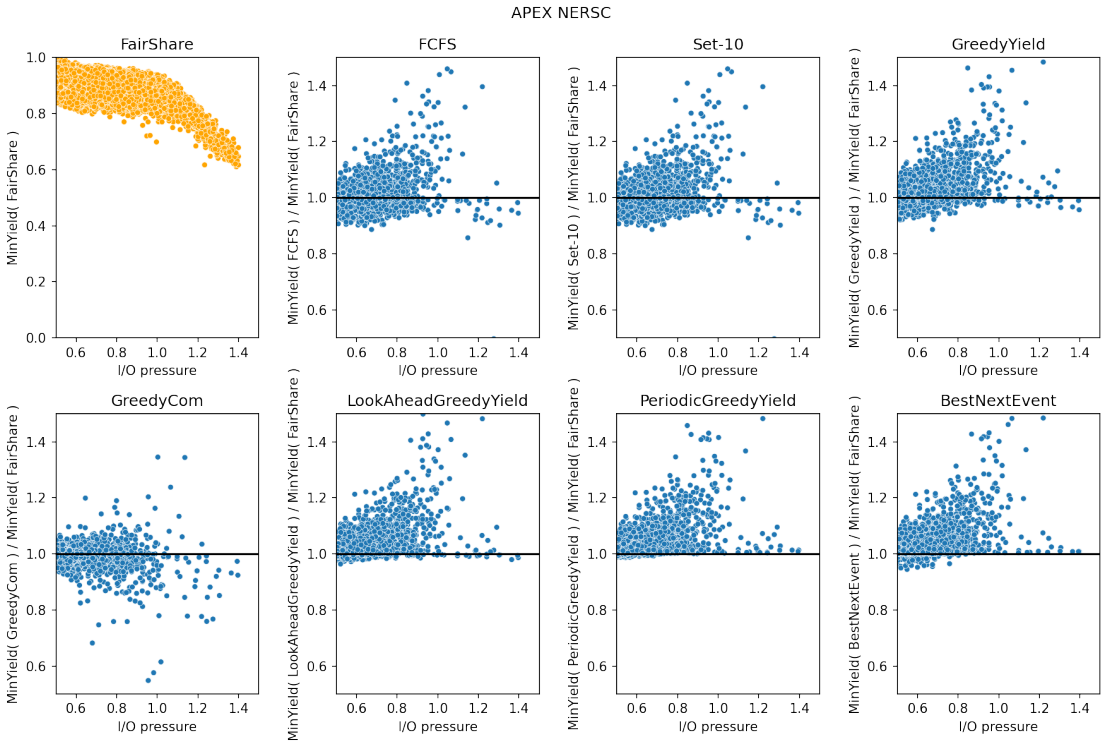


Figure 7.15: MinYield of all strategies, as a ratio of the MinYield with the FairShare strategy for the same experiment, for the NERSC workload, as a function of the I/O Pressure.

an independent graph per strategy. As a reference, the MinYield of FairShare is also presented in a different color. A value of 1 of the ratios means that the target strategy obtains the exact same MinYield as FairShare for the scenario, while a value higher than 1 means that a higher MinYield than FairShare is obtained for this scenario, and a value lower than 1 that on this scenario, the strategy obtains a lower MinYield than FairShare.

There are three classes of graphs in this figure. The strategy GreedyCom presents on average a ratio distributed approximately uniformly between 0.9 and 1.1. This means that this strategy fails to reliably improve the MinYield in at least half of the scenarios. The second set of graphs show that FCFS, Set-10, and GreedyYield have a non-negligible set of scenarios where they decrease MinYield compared to FairShare, but as the I/O pressure increases they tend to behave slightly better than MinYield on average (with still a risk of significant performance degradation for all I/O pressures). When they experience gains, the gains are more pronounced for high I/O pressures. Set-10 and FCFS behave strictly identically over the NERSC workload, and this is because the NERSC workload featuring very small windows with typically at most one phase for many applications, Set-10 does not have time to learn the phases, and thus puts all the applications in the same set, behaving as FCFS.

The third set of graphs include LookAheadGreedyYield, PeriodicGreedyYield, and BestNextEvent. These three strategies have a very high probability of increasing MinYield compared to FairShare, and that performance increase tends to be higher as the I/O pressure increases. BestNextEvent is the strategy of this set that features the highest risk of decreasing MinYield (although the decrease is limited to 95% of the

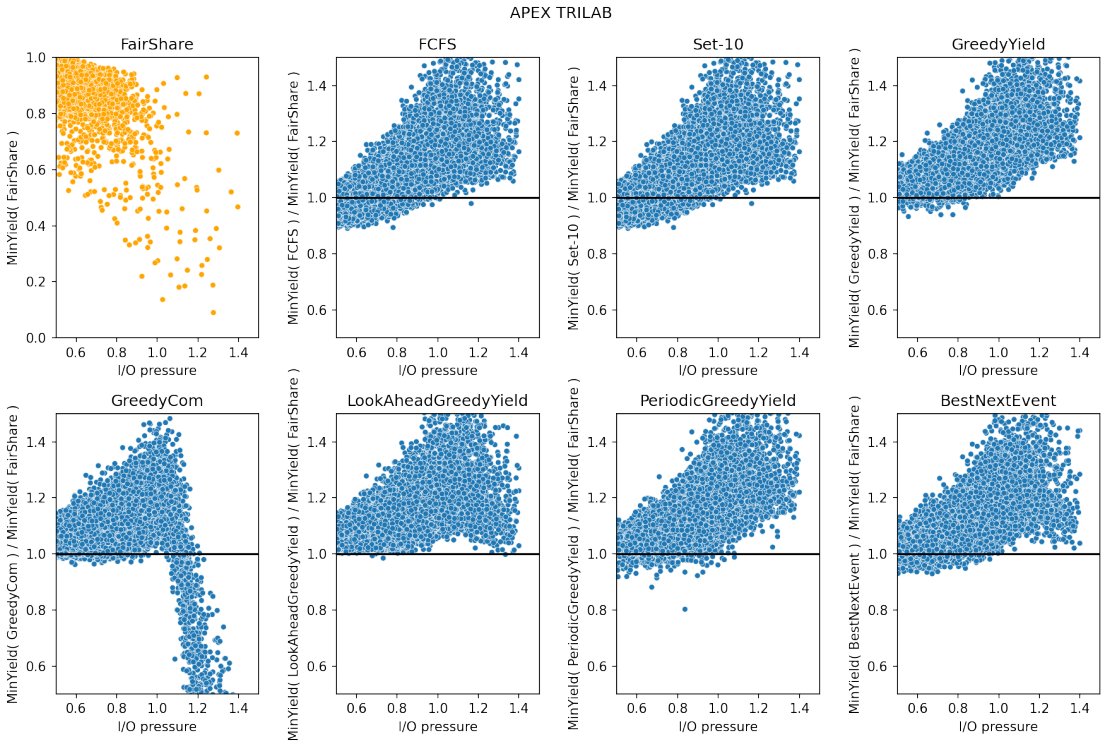


Figure 7.16: MinYield of all strategies, as a ratio of the MinYield with the FairShare strategy for the same experiment, for the TRILAB workload, as a function of the I/O pressure.

MinYield of FairShare in the worst scenario), while PeriodicGreedyYield has almost no scenario with a MinYield lower than FairShare.

MinYield of All Strategies on APEX TRILAB Scenarios

Figure 7.16 presents the same evaluation, for the TRILAB workload (relative to the experiments shown in the right graph of Figure 7.14). With this workload, the ratio of MinYield behaves differently than with the NERSC workload. Overall, all strategies tend to behave better (with relatively less scenarios presenting a ratio lower than 1), and the gains over FairShare are on average higher for all strategies at low I/O pressure and for most strategies at high I/O pressure.

GreedyCom presents better behaviors than over the NERSC workload, with only a few scenarios underperforming FairShare, until the I/O pressure reaches a ratio of 1, i.e., until the system reaches saturation of the communication system. Then, the performance of GreedyCom quickly drops dramatically, with eventually all scenarios obtaining a lower MinYield than FairShare.

FCFS, Set-10 and GreedyYield continue to behave similarly, but the trend is more clear, with a significant risk of MinYield degradation for low I/O pressures, but significant gains as the I/O pressure, and consistent gains at I/O saturation (when the I/O pressure is higher than 1). FCFS and Set-10 continue to behave identically. However, this time this is not due to a lack of time to learn the periodicity of the applications: in the TRILAB workload, each application has a minimum of 5 phases during the window, which is long enough to converge on the phase duration and categorize the application in the appropriate set. Because all applications checkpoint with the same approximate checkpointing period,

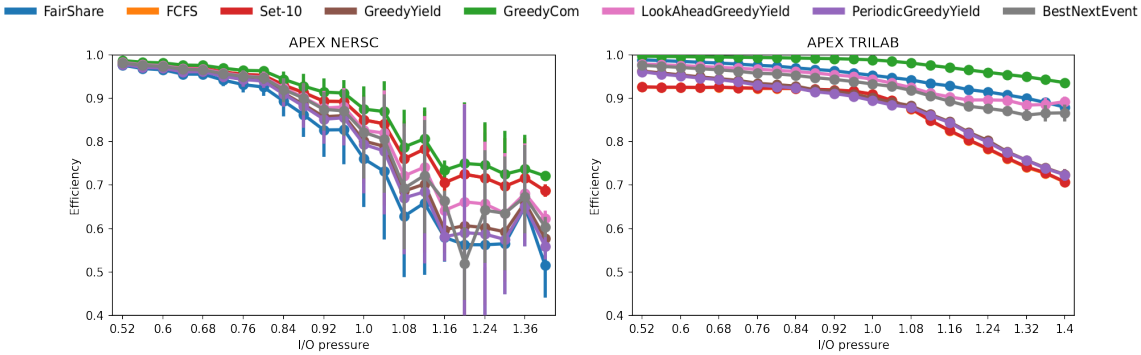


Figure 7.17: EFFICIENCY of all strategies for the NERSC and TRILAB workloads, as a function of the I/O pressure.

only the duration of the checkpoint operation can define different categories of phases. The checkpoint duration of the different applications can vary by an order of magnitude or more in the TRILAB workload, but the duration of the slowest checkpointing operation still remains small compared to the checkpointing period. As a consequence, the SET-10 strategy tends to put all applications in the same category, and falls back to applying the FCFS strategy.

Among the three wining strategies for the NERSC workload, PERIODICGREEDYYIELD, LOOKAHEADGREEDYYIELD and BESTNEXTEVENT, the trends observed for the NERSC workload are enforced: until the system reaches I/O saturation, PERIODICGREEDYYIELD and BESTNEXTEVENT feature a few scenarios where MINYIELD can be slightly decreased compared to FAIRSHARE, but in most scenarios (and in almost all scenarios for LOOKAHEADGREEDYYIELD), these strategies improve MINYIELD, and that improvement becomes higher as the I/O pressure increases. Contrarily to NERSC traces, GREEDYYIELD performs similarly to PERIODICGREEDYYIELD and BESTNEXTEVENT on the TRILAB traces.

On these longer windows, the I/O pressure seems to have a more significant impact than on the smaller windows of the NERSC workload, and as the I/O pressure increases, the gains relative to FAIRSHARE tend to increase (see paragraph "Note on APEX traces" in the same section below for detailed data on the window's length). When the I/O pressure is higher than 1, interference is unavoidable, and the I/O scheduling strategy becomes critical to the performance of applications. Naive strategies, or strategies that are not well suited for the irregular nature of the applications present in these workloads, have then a higher risk of taking the wrong decision and performing worse than FAIRSHARE.

Efficiency of All Strategies on APEX Scenarios

Figure 7.17 presents the mean and standard deviation of the EFFICIENCY metric for each strategy as a function of the I/O pressure. To synthetize these graphs, we split the I/O pressure domain in 25 intervals and compute the mean EFFICIENCY value and its standard deviation for all scenarios with an I/O pressure that falls in this interval. The point is presented at the middle of the interval.

The NERSC and TRILAB workloads present both some commonalities and some significantly different features. In the NERSC workload, EFFICIENCY quickly drops as the I/O pressure increases for all strategies, while each strategy seems to hold its EFFICIENCY until the system reaches saturation (I/O pressure of 1) in the TRILAB workload. Once the I/O pressure is above 1, EFFICIENCY drops with the I/O pressure for both workloads, but

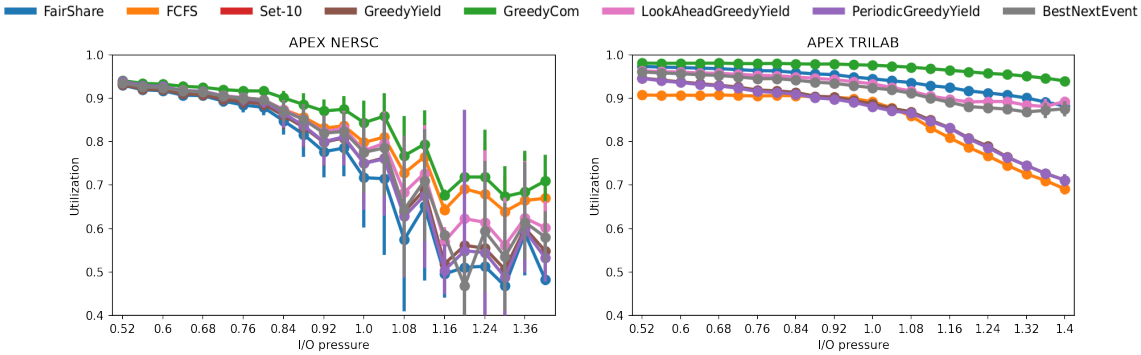


Figure 7.18: UTILIZATION of all strategies for the NERSC and TRILAB workloads, as a function of the I/O Pressure.

this drop is more pronounced, and becomes chaotic, for the NERSC workload, while the EFFICIENCY with the TRILAB workload remains stable and supports higher I/O pressures for all strategies.

EFFICIENCY measures the sum of actual progress of all applications throughout the window. As NERSC has on average much smaller windows than TRILAB, the effect of a few bad I/O schedule decisions can be much more impactful on a small window than on a large one. This explains the chaotic EFFICIENCY measurement on the NERSC workload compared to TRILAB.

TRILAB is also a workload on which it is easier, for all strategies, to maintain a high EFFICIENCY compared to NERSC, because the windows feature a lower number of long and large-scale applications, where the I/O is close to periodic per application (mostly driven by fixed-period checkpointing), allowing many opportunities to overlap I/O operations and computation. However, at high I/O pressure, we observe three groups of strategies on the TRILAB workload: GREEDYCOM, which targets a balance of I/O operation progress, remains the most efficient; FAIRSHARE, LOOKAHEADGREEDYYIELD and BESTNEXTEVENT provide a similar EFFICIENCY, slightly under GREEDYCOM; and in the third group, SET-10, FCFS (hidden by SET-10 in the figure), GREEDYYIELD (hidden by PERIODICGREEDYYIELD in the figure), and PERIODICGREEDYYIELD present the worst EFFICIENCY. As the I/O pressure is above 1, contentions are unavoidable, and the strategies that pursue too eagerly an optimization of MINYIELD fail at providing a good EFFICIENCY. FCFS and SET-10 take I/O scheduling decisions that are detrimental to EFFICIENCY because the I/O that is favored is arbitrary.

In the NERSC workload, the metric is too chaotic at high I/O pressure to define a clear order, but GREEDYCOM remains the strategy with the highest EFFICIENCY, which is expected as GREEDYCOM targets this metric.

Utilization of All Strategies on APEX Scenarios

Figure 7.18 presents the mean and standard deviation of the UTILIZATION metric for each strategy as a function of the I/O pressure. We used the same binning approach as for Figure 7.17 to present trends from individual scenarios.

UTILIZATION is overall lower in the NERSC workload than in the TRILAB workload. This is corroborated by the window characteristics detailed in paragraph "Notes on Apex Traces" in the same section below: windows in the NERSC workload have on average a lower UTILIZATION than for the TRILAB workload, even without considering I/O interferences.

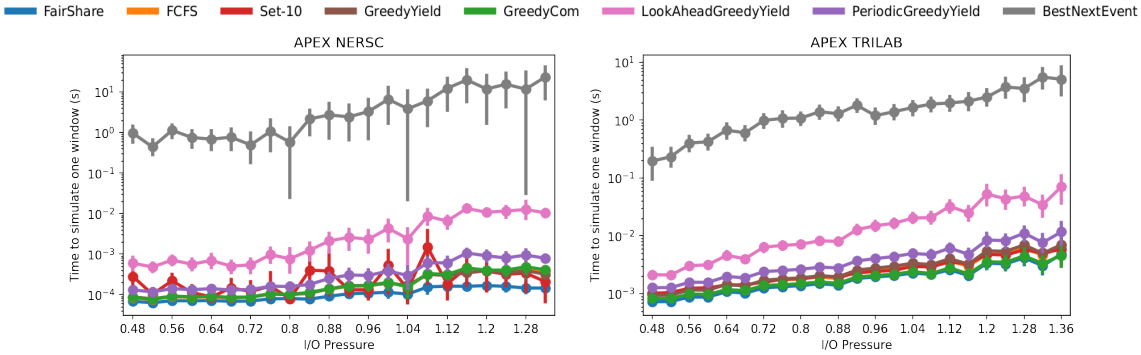


Figure 7.19: Simulation time of all strategies for the NERSC and TRILAB workloads, as a function of the I/O Pressure.

As the I/O pressure increases and in the saturated domain in particular, I/O interferences reduce even more UTILIZATION, for all strategies and in all scenarios. GREEDYCOM, which targets a balance of I/O operation progress, shines with this metric as well as for EFFICIENCY, at the cost of a worst MINYIELD as illustrated in Figures 7.15 and 7.16. On these practical scenarios, EFFICIENCY and UTILIZATION seem to behave very similarly.

Computation Time of All Strategies on APEX Scenarios

Last, we look at the computation time of the different strategies. Each strategy decides to take a scheduling decision at different times, and each scheduling decision impacts the order of events and when the next scheduling decision will happen. To compare the computation time of the different strategies in a practical setup, we have thus chosen to measure the entire simulation time of a given window, for a given strategy. This time includes the simulation, but also, for each scheduling event, the cost of computing the decision, as an implementation of the strategy would have to do.

The mean and standard deviation of the time to simulate each of the windows is presented in Figure 7.19. We used the same binning approach as for Figures 7.17 and 7.18, in order to present trends. As the different strategies exhibit very different simulation computation times, the time axis is using a logarithmic scale.

BESTNEXTEVENT is the only strategy that requires significant computation time, with a few seconds (and never more than 60 seconds) needed to simulate an entire window for both the NERSC and TRILAB workloads. The second highest demanding strategy, LOOKAHEADGREEDYYIELD, only requires 10s of milliseconds to simulate an entire window, and all the other strategies are yet an order of magnitude faster.

Although BESTNEXTEVENT is the most demanding strategy in terms of computational complexity, its runtime remains small enough to be considered in practice. The PERIODICGREEDYYIELD strategy, which needs to re-compute regularly the entire schedule, can be called with a very small period (seconds to milliseconds), as its computational demand on realistic scenarios is achieved in a fraction of this time.

Note on APEX traces

To understand the results obtained on the APEX traces, we look at the characteristics of the TRILAB and NERSC workloads on the different projected machines. Figure 7.20 presents the distribution of the window duration for both workloads, and for 11 projected machines (Celio, and M_1 to M_{10}). We observe that window duration is an order of magnitude longer

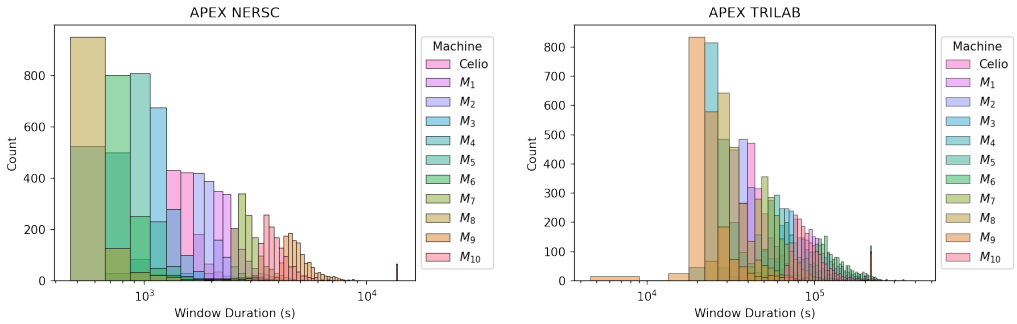


Figure 7.20: Distribution of the window durations for the APEX campaign on the NERSC and TRILAB workloads, as function of the projected machine.

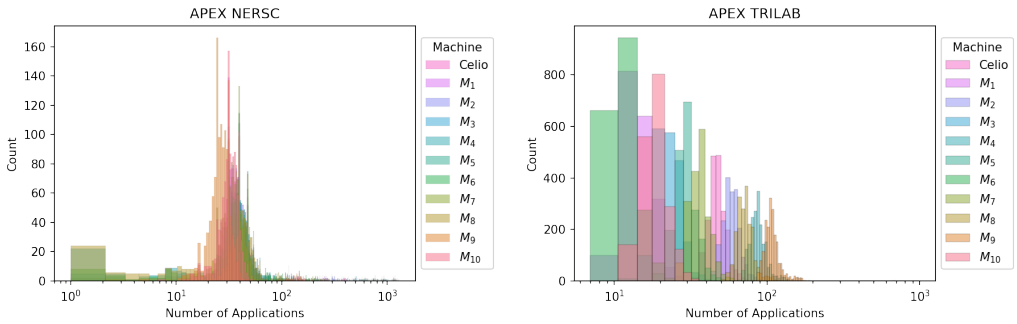


Figure 7.21: Distribution of the number of applications within each window for the APEX campaign on the NERSC and TRILAB workloads, as function of the projected machine.

for the TRILAB workload compared to the NERSC workload, for all machines. This is easily explained by the difference of jobs between the workloads: the NERSC workload features a large number of short-lived applications, it is thus hard to find long time windows during which no application completes or starts, compared to the TRILAB workload. We also observe that, as the platform becomes larger, the average window duration increases for both workloads. Problem size, and thus amount of I/O, is defined as a function of the aggregated memory in the APEX report. When we scale up the number of nodes to project a future machine, we maintain a memory of 1GB per core, but as the number of cores per node increases, the amount of memory per application increases. This results in longer-running applications, and thus in longer windows.

These observations are corroborated by Figure 7.21. This figure observes how many applications belong to a given window, for the different machines. The TRILAB workload presents windows that have an order of magnitude fewer applications than the NERSC workload, which corresponds well to the relative workload distribution of the different workloads.

Figure 7.22 presents the distribution of platform usage during a simulation window for both workloads. This metric varies significantly, but is clearly higher for the TRILAB workload in average than what is observed for the NERSC platform. Again, the small duration of some applications in the NERSC workload introduces this imbalance: selecting long windows in order to provide a statistically accurate result favors windows that have only a few occurences of the short-lived application, which increases the probability that some nodes are left idle by the first-fit scheduler. This low utilization translates in a lower I/O pressure, which explains why the NERSC workload shows less points in the high I/O

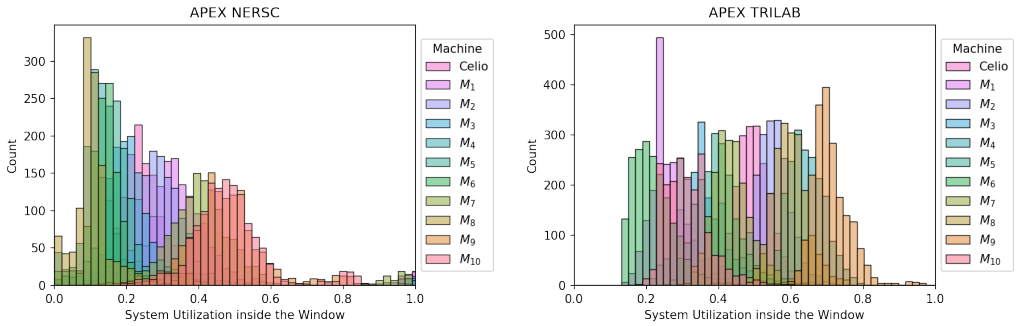


Figure 7.22: Distribution of the number of applications within each window for the APEX campaign on the NERSC and TRILAB workloads, as function of the projected machine.

pressure end of the figures.

Synthesis of the Evaluation on APEX Scenarios

Overall, LookAheadGreedyYield is the strategy that shows the best performance for the MinYield metric on the most variety of scenarios, closely followed by PeriodicGreedyYield and BestNextEvent. PeriodicGreedyYield requires to re-compute goals at a higher frequency, namely twice the frequency of the other greedy strategies with our choice for the periodicity of external events; but LookAheadGreedyYield remains more costly, because each decision requires a set of goal computations, one per active application, and BestNextEvent, with its exhaustive search, is far more computationally demanding. GreedyCom is a strategy that would perform the best on the Utilization and Efficiency metrics, and its MinYield remains reasonable on the TRILAB workload, as long as the I/O pressure is not saturated, but it is a risky choice for the NERSC workload.

7.7 Conclusion

This work has revisited I/O bandwidth-sharing strategies for concurrent applications. Our main contributions are two-fold. On the theoretical side, we have provided the first competitive ratios for such strategies, owing to a rigorous framework based upon steady-state windows. These competitive ratios are mostly negative. In particular, the lower bound for MinYield is as high as the (order of) number of applications for all strategies except PeriodicGreedyYield. These results bring new insights on the hardness of the problem, and lay the foundations for the study of its complexity. On the practical side, we have introduced several new greedy heuristics and have compared them to well-established strategies such as FCFS, FairShare and Set-10. We have used a comprehensive set of experiments, some based upon synthetic traces and some based upon an extended version of APEX traces. In both cases, the well-established strategies perform worse, and often much worse, than the new heuristics. As a global conclusion, although there is no absolute winner for all scenarios and objectives, we recommend using LookAheadGreedyYield, which achieves an excellent performance for MinYield on all scenarios, achieves better Efficiency and Utilization than FairShare for the NERSC workload and comparable ones for the TRILAB and synthetic workloads. LookAheadGreedyYield requires knowing the volume of an I/O operation when it is posted. If such an information is not available,

one can use PeriodicGreedyYield: it achieves very good MinYield on all scenarios, achieves better Efficiency and Utilization than FairShare for the NERSC workload, comparable ones for the synthetic workload, but worse ones for the TRILAB workload. We recommend that the I/O community would implement LookAheadGreedyYield and PeriodicGreedyYield for further assessment.

Chapter 8

Conclusion and future work

Throughout this thesis, we have delved into the critical issue of resilience in large-scale computing systems. The rapid development of high-performance computing has brought forth new challenges, and one of the most pressing among them is the need for efficient and robust fault-tolerance mechanisms. Our research has sought to address this challenge by focusing on the optimization of checkpointing strategies, the analysis of various resilience techniques, and the development of novel scheduling approaches for handling failed jobs in batch schedulers.

Over the course of our investigation, we have made several contributions to the field. These contributions have provided valuable insights into the design and implementation of fault-tolerance mechanisms, helping to improve the efficiency and reliability of large-scale scientific computations. In addition, our research has shed light on the performance trade-offs associated with different resilience techniques, offering guidance for researchers and practitioners seeking to develop effective strategies for dealing with failures in large-scale computing systems.

One important aspect of our work has been the exploration of the impact of different failure models on the performance of checkpointing strategies. By considering non-memoryless failure distributions, we have been able to develop more accurate models of real-world computing systems, leading to more effective checkpointing strategies that can provide substantial performance improvements. Moreover, we have shown that our proposed strategies are applicable not only to single parallel applications but also to workflows composed of parallel tasks with dependencies, thereby further expanding their potential impact.

We have also advanced the state of the art in the design of scheduling algorithms that account for failures and performance variations. We have developed online risk-aware scheduling strategies that adapt to variable capacity environments and provide better performance than traditional first-fit algorithms. These strategies have demonstrated their value in real-world settings, where performance variations and failures are commonplace, and have shown potential for further refinement and improvement.

In addition to these primary areas of focus, our work has also touched upon the challenges associated with I/O bandwidth-sharing in concurrent applications. We have presented novel greedy heuristics that outperform well-established strategies, and we have provided the first competitive ratios for these strategies, laying the groundwork for further study of the problem complexity.

Future research could explore several directions. More specifically, we first detail direct extensions of the work presented in this thesis. In Chapter 2, we have studied the problem of scheduling moldable parallel jobs to cope with silent errors. We presented a formal

model and designed two resilient scheduling algorithms. The results demonstrated their practical usefulness and robustness under common job speedups and parameter settings. Future work includes investigating alternative failure models and exploring checkpointing and rollback recovery for long-running jobs.

In Chapter 3, we focused on the online scheduling of moldable task graphs to minimize makespan under various speedup models. We designed a new online algorithm and derived competitive ratios for several other speedup models. Future work involves extending the algorithm and analysis to other common speedup models, online scheduling settings, and evaluating the performance of our algorithm using realistic workflows.

In Chapter 4, we presented online risk-aware scheduling strategies to preserve performance in variable capacity environments. Our assessment using workload traces and synthetic traces showed significant gains over first-fit algorithms. Future work includes exploring different workloads, job execution models, variation models, and malleable workloads.

In Chapter 5, we investigated checkpointing strategies to protect parallel applications from non-memoryless failures. We designed a general strategy, NextStep, which maximizes the expected efficiency until the next failure. Our extensive simulation results showed the significant impact and superiority of NextStep compared to traditional solutions.

In Chapter 6, we investigated checkpointing strategies for parallel workflows, assuming that tasks can be checkpointed at any instant. We introduced CheckMore strategies that may checkpoint some tasks more often than others, and more often than in the MinExp strategy. Our simulations demonstrated the necessity of checkpointing workflow tasks more often than with the classical Young/Daly strategy. Future work includes extending the simulation campaign to parallel tasks and investigating the impact of the failure-free list schedule on the final performance.

In Chapter 7, we revisited I/O bandwidth-sharing strategies for concurrent applications. We provided the first competitive ratios for such strategies and introduced several new greedy heuristics. Our experiments showed that the new heuristics outperformed well-established strategies. We recommend using PeriodicGreedyYield for further assessment.

Altogether, this thesis has made contributions to the understanding and improvement of resilience in high-performance computing systems. By developing innovative checkpointing strategies, scheduling algorithms, and I/O bandwidth-sharing techniques, we have laid the foundation for more efficient, reliable, and scalable computing systems in the future. Since the field of high-performance computing continues to evolve, the insights and methods gained from our research will serve as a valuable resource for researchers and practitioners working to address the ever-present challenge of resilience.

In the long term, a promising research direction is the integration of multiple fault-tolerance mechanisms, such as combining checkpointing with replication or erasure coding, to provide even greater resilience in the presence of failures. Additionally, the development of adaptive and self-tuning fault-tolerance strategies that can respond to changing system conditions and application requirements could lead to even more efficient and robust computing systems.

Bibliography

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, 2010.

[2] B. Aksanli, J. Venkatesh, L. Zhang, and T. Rosing. Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, New York, NY, USA, 2011. Association for Computing Machinery.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'67*, volume 30, pages 483–485. AFIPS Press, 1967.

[4] Anonymous. Extended version of the ICPP submission, Apr. 2023. https://zenodo.org/record/7813175.

[5] Argonne Leadership Computing Facility. Mira log traces. https://reports.alcf.anl.gov/data/mira.html.

[6] M. Atkinson, S. Gesing, J. Montagnat, and I. Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216–227, 2017.

[7] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. Scheduling computational workflows on failure-prone platforms. *Int. J. of Networking and Computing*, 6(1):2–26, 2016.

[8] G. Aupy, A. Gainaru, and V. Le Fèvre. Periodic I/O scheduling for super-computers. In S. A. Jarvis, S. A. Wright, and S. D. Hammond, editors, *PMBS workshop*, volume 10724 of *Lecture Notes in Computer Science*, pages 44–66. Springer, 2017.

[9] G. Aupy, A. Gainaru, and V. Le Fèvre. I/O scheduling strategy for periodic applications. *ACM Trans. Parallel Comput (TOPC)*, 6(2):1–26, 2019.

[10] G. Aupy, Y. Robert, and F. Vivien. Assuming failure independence: are we right to be wrong? In *FTS'2017, the Workshop on Fault-Tolerant Systems, in conjunction with Cluster'2017*. IEEE Computer Society Press, 2017.

[11] F. Azzedin and M. Maheswaran. Integrating trust into grid resource management systems. In *31st International Conference on Parallel Processing (ICPP)*, pages 47–54. IEEE Computer Society, 2002.

[12] P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: A polynomial time algorithm for offline dynamic power management. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, page 364–367, USA, 2006. Society for Industrial and Applied Mathematics.

[13] J. Basney and M. Livny. Improving goodput by coscheduling cpu and network capacity. *The International Journal of High Performance Computing Applications*, 13(3):220–230, 1999.

[14] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir. Reducing waste in extreme scale systems through introspective analysis. In *IPDPS*, pages 212–221. IEEE, 2016.

[15] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proc. SC'11*, 2011.

[16] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *ICPP*, pages 72–75, 1990.

[17] K. P. Belkhale, P. Banerjee, and W. S. Av. A scheduling algorithm for parallelizable dependent tasks. In *IPPS*, pages 500–506, 1991.

[18] A. Beloglazov and R. Buyya. Energy efficient resource management in virtualized cloud data centers. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831, 2010.

[19] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, SODA'02, page 762–771. SIAM, 2002.

[20] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. Towards optimal multi-level checkpointing. *IEEE Trans. Computers*, 66(7):1212–1226, 2017.

[21] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Trans. Parallel Computing*, 3(2), 2016.

[22] A. Benoit, Y. Du, T. Herault, L. Marchal, G. Pallez, L. Perotin, Y. Robert, H. Sun, and F. Vivien. Checkpointing à la Young/Daly: an overview. In *IC3, the 14th Int. Conf. on Contemporary Computing*. ACM Press, 2022.

[23] A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, and H. Sun. Resilient scheduling of moldable jobs on failure-prone platforms. In *IEEE Cluster*, 2020.

[24] A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, and H. Sun. Resilient scheduling of moldable parallel jobs to cope with silent errors. *IEEE Transactions on Computers*, 71(07):1696–1710, 2022.

[25] A. Benoit, V. Le Fèvre, P. Raghavan, Y. Robert, and H. Sun. Design and comparison of resilient scheduling heuristics for parallel jobs. In *APDCM*, 2020.

[26] A. Benoit, L. Perotin, Y. Robert, and H. Sun. Online scheduling of moldable task graphs under common speedup models. In *ICPP*, pages 51:1–51:11, 2022.

[27] V. Bharadwaj, D. Ghose, and T. Robertazzi. Divisible load theory: a new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[28] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[29] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In *Euro-Par*, pages 191–197, 2001.

[30] F. Boito, G. Pallez, L. Teylo, and N. Vidal. IO-SETS: Simple and efficient approaches for I/O bandwidth management. working paper or preprint, May 2022.

[31] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of SC'11*, 2011.

[32] P. Brucker, S. Knust, and C. Oğuz. Scheduling chains with identical jobs and constant delays on a single machine. *Mathematical Methods of Operations Research*, 63(1):63–75, 2006.

[33] K. W. Cameron, R. Ge, and X. Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, 2005.

[34] L. Canon, L. Marchal, B. Simon, and F. Vivien. Online scheduling of task graphs on heterogeneous platforms. *IEEE Trans. Parallel Distributed Syst.*, 31(3):721–732, 2020.

[35] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman. System-level scalable checkpoint-restart for petascale computing. In *22nd Int. Conf.Parallel and Distributed Systems (ICPADS)*. IEEE, 2016.

[36] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[37] F. Cappello, K. Mohror, et al. VeloC: very low overhead checkpointing system. https://veloc.readthedocs.io/en/latest/, Mar. 2019.

[38] J. Carretero, E. Jeannot, G. Pallez, D. E. Singh, and N. Vidal. Mapping and scheduling HPC applications for optimizing I/O. In *ICS: International Conference on Supercomputing*. ACM, 2020.

[39] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[40] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 297–305. ACM, 2002.

[41] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande. LADR: Low-cost application-level detector for reducing silent output corruptions. In *HPDC*, pages 156–167, 2018.

[42] C.-Y. Chen and C.-P. Chu. A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints. *IEEE Trans. Parallel Distrib. Syst.*, 24(8):1479–1488, 2013.

[43] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. *SIGPLAN Not.*, 48(8):167–176, 2013.

[44] M. Chrobak and C. Kenyon-Mathieu. Sigact news online algorithms column 10: Competitiveness via doubling. *SIGACT News*, 37(4):115–126, 2006.

[45] K. L. Chung. *A Course in Probability Theory*. Stanford University, 3 edition, 2000.

[46] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Inf.*, 1(3):200–213, 1972.

[47] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[48] G. Demirci, H. Hoffmann, and D. H. K. Kim. Approximation algorithms for scheduling with resource and precedence constraints. In *STACS*, pages 25:1–25:14, 2018.

[49] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *IPDPS*. IEEE, 2014.

[50] S. Di, H. Guo, R. Gupta, E. R. Pershey, M. Snir, and F. Cappello. Exploring properties and correlations of fatal events in a large-scale HPC system. *Trans. on Parallel and Distributed Systems*, 2018.

[51] S. Di, Y. Robert, F. Vivien, and F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model. *IEEE Trans. Parallel & Distributed Systems*, 2016, preprint available on the IEEE digital library.

[52] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *Proc. 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, page 155–164. IEEE Computer Society, 2014.

[53] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, 2(4):473–487, 1989.

[54] R. A. Dutton and W. Mao. Online scheduling of malleable parallel jobs. In *PDCS*, pages 136–141, 2007.

[55] K. Ecker and M. Tanaś. Complexity of scheduling of coupled tasks with chains precedence constraints and any constant length of gap. *Journal of the Operational Research Society*, 63(4):524–529, 2012.

[56] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *43rd Int. Conf. Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[57] N. El-Sayed and B. Schroeder. To checkpoint or not to checkpoint: Understanding energy-performance-i/o tradeoffs in hpc checkpointing. In *CLUSTER*, pages 93–102, 2014.

[58] Fault-Tolerance Research Hub. User level failure mitigation, 2021. https://fault-tolerance.org.

[59] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[60] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal on-line scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[61] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *SC'11*. ACM, 2011.

[62] R. Ferreira da Silva, L. Pottier, T. a. Coleman, E. Deelman, and H. Casanova. Workflowhub: Community framework for enabling scientific workflow research and development. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 49–56, 2020.

[63] U. S. N. S. Foundation. Design for environmental sustainability in computing (desc), Dec. 2022. https://beta.nsf.gov/funding/opportunities/design-environmental-sustainability-computing-desc.

[64] A. Frank, M. Baumgartner, R. Salkhordeh, and A. Brinkmann. Improving checkpointing intervals by considering individual job failure probabilities. In *IPDPS*, pages 299–309, 2021.

[65] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the I/O of HPC applications under congestion. In *IPDPS'2015, the 29th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2015.

[66] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[67] E. Gelenbe, P. Boryszko, M. Siavvas, and J. Domanska. Optimum checkpoints for time and energy. In *28th MASCOTS*, pages 1–8. IEEE, 2020.

[68] I. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini. Greenslot: Scheduling energy consumption in green datacenters. In *SC '11: Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[69] Google team. Borg cluster workload traces, 2019. https://github.com/google/cluster-data.

[70] P. J. Grabner and H. Prodinger. Maximum statistics of n random variables distributed by the negative binomial distribution. *Combinatorics, Probability and Computing*, 6(2):179–183, 1997.

[71] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[72] Green ICT. Follow the sun, wind, moon, 2009. https://www.vertatique.com/cloud-computing-starting-follow-sunwindmoon.

[73] A. Guermouche, L. Marchal, B. Simon, and F. Vivien. Scheduling trees of malleable tasks for sparse linear algebra. In *Euro-Par*, pages 479–490, 2015.

[74] P.-L. Guhur, H. Zhang, T. Peterka, E. Constantinescu, and F. Cappello. Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In *Euro-Par*, 2016.

[75] U. Gupta, M. Elgamal, G. Hills, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu. Act: Designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 784–799, New York, NY, USA, 2022. Association for Computing Machinery.

[76] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien. Checkpointing workflows for fail-stop errors. *IEEE Trans. Computers*, 67(8):1105–1120, 2018.

[77] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. A generic approach to scheduling and checkpointing workflows. In *ICPP'2018, the 47th Int. Conf. on Parallel Processing*. IEEE Computer Society Press, 2018.

[78] J. T. Havill and W. Mao. Competitive online scheduling of perfectly malleable jobs with setup times. *European Journal of Operational Research*, 187:1126–1142, 2008.

[79] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proc. SC'11*, 2011.

[80] T. Herault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[81] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K. B. Ferreira, G. Bosilca, and J. Dongarra. Checkpointing strategies for shared high-performance computing platforms. *International Journal of Networking and Computing*, 9(1):28–52, 2019.

[82] S. Hiroyama, T. Dohi, and H. Okamura. Aperiodic checkpoint placement algorithms—survey and comparison. *Journal of Software Engineering and Applications*, 6(4A):41–53, 2013.

[83] U. Hönig and W. Schiffmann. A parallel branch-and-bound algorithm for computing optimal task graph schedules. In *Second International Workshop on Grid and Cooperative Computing GCC*, pages 18–25, 2003.

[84] T. C. Hu. Parallel sequencing and assembly line problems. *Oper. Res.*, 9(6):841–848, 1961.

[85] Y. Hua, X. Shi, H. Jin, W. Liu, Y. Jiang, Y. Chen, and L. He. Software-defined qos for I/O in exascale computing. *CCF Trans. High Perform. Comput.*, 1(1):49–59, 2019.

[86] J. L. Hurink and J. J. Paulus. Online algorithm for parallel job scheduling and strip packing. In C. Kaklamanis and M. Skutella, editors, *Approximation and Online Algorithms*, pages 67–74. Springer, 2008.

[87] IBM Spectrum LSF Job Scheduler. Fault tolerance and automatic management host failover, 2021. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=cluster-fault-tolerance.

[88] F. Isaila, J. Carretero, and R. Ross. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *16th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 346–355. IEEE, 2016.

[89] K. Jansen. A $(3/2 + \epsilon)$ approximation algorithm for scheduling moldable and non-moldable parallel tasks. In *SPAA*, pages 224–235, 2012.

[90] K. Jansen and F. Land. Scheduling monotone moldable jobs in linear time. In *IPDPS*, pages 172–181, 2018.

[91] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA*, pages 490–498, 1999.

[92] K. Jansen and R. Thöle. Approximation algorithms for scheduling parallel jobs. *SIAM Journal on Computing*, 39(8):3571–3615, 2010.

[93] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *SPAA*, page 86–95, 2005.

[94] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Trans. Algorithms*, 2(3):416–434, 2006.

[95] E. Jeannot, G. Pallez, and N. Vidal. Scheduling periodic I/O access with bi-colored chains: models and algorithms. *J. Scheduling*, 24(5):469–481, 2021.

[96] B. Johannes. Scheduling parallel jobs to minimize the makespan. *J. of Scheduling*, 9(5):433–452, 2006.

[97] T. Johnson, T. A. Davis, and S. M. Hadfield. A concurrent dynamic task graph. *Parallel Computing*, 22(2):327–333, 1996.

[98] N. Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, Sept. 2018.

[99] W. Jones, J. Daly, and N. DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *HPDC'10*, pages 276–279. ACM, 2010.

[100] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao. Improving MPI collective i/o for high volume non-contiguous requests with intra-node aggregation. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2682–2695, 2020.

[101] A. Kassab, J.-M. Nicod, L. Philippe, and V. Rehn-Sonigo. Scheduling independent tasks in parallel under power constraints. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 543–552, 2017.

[102] N. Kell and J. Havill. Improved upper bounds for online malleable job scheduling. *J. of Scheduling*, 18(4):393–410, 2015.

[103] O. Kella and W. Stadje. Superposition of renewal processes and an application to multi-server queues. *Statistics & probability letters*, 76(17):1914–1924, 2006.

[104] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim. An analysis of system balance and architectural trends based on top500 supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2021, page 11–22, New York, NY, USA, 2021. Association for Computing Machinery.

[105] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10. ACM, 2010.

[106] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM J. Comput.*, 30(1):300–317, Apr. 2000.

[107] Y. Kuo, S.-I. Chen, and Y.-H. Yeh. Single machine scheduling with sequence-dependent setup times and delayed precedence constraints. *Operational Research*, 20(2):927–942, 2020.

[108] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[109] LANL, NERSC, SNL. APEX workflows. Technical report, Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratory (SNL)., 2016. Available online at http://www.nersc.gov/assets/apex-workflows-v2.pdf.

[110] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. In *ESA*, pages 146–157, 2001.

[111] S. Levy and K. B. Ferreira. An examination of the impact of failure distribution on coordinated checkpoint/restart. In *FTXS Workshop*, pages 35–42. ACM, 2016.

[112] K. Li. Analysis of the list scheduling algorithm for precedence constrained parallel tasks. *Journal of Combinatorial Optimization*, 3(1):73–88, 1999.

[113] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Trans. on computers*, pages 699–708, 2001.

[114] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice and Experience*, 40(11):943–964, 2010.

[115] L. Lozano, M. J. Magazine, and G. G. Polak. Decision diagram-based integer programming for the paired job scheduling problem. *IISE Transactions*, 53(6):671–684, 2021.

[116] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA*, pages 167–176, 1994.

[117] M. Maiterth, G. Koenig, K. Pedretti, S. Jana, N. Bates, A. Borghesi, D. Montoya, A. Bartolini, and M. Puzovic. Energy and power aware job scheduling and resource management: Global survey—initial analysis. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 685–693. IEEE, 2018.

[118] M. Maiterth, T. Wilde, D. Lowenthal, B. Rountree, M. Schulz, J. Eastep, and D. Kranzlmüller. Power aware high performance computing: Challenges and opportunities for application and system developers—survey & tutorial. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 3–10. IEEE, 2017.

[119] Marc Snir et al. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, 2014.

[120] Message Passing Interface Forum. MPI: A message-passing interface standard, version 4.0. https://www.mpi-forum.org/, 2021.

[121] R. H. Möhring, A. S. Schulz, and M. Uetz. Approximation in stochastic scheduling: The power of LP-based priority policies. *J. ACM*, 46(6):924–942, 1999.

[122] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. SC'10*, 2010.

[123] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA*, pages 23–32, 1999.

[124] G. Mounié, C. Rapine, and D. Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Comput.*, 37(2):401–412, 2007.

[125] A. Munier, M. Queyranne, and A. S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In *Integer Programming and Combinatorial Optimization*, pages 367–382. Springer, 1998.

[126] A. Munier and F. Sourd. Scheduling chains on a single machine with non-negative time lags. *Mathematical Methods of Operations Research*, 57(1):111–123, 2003.

[127] National Energy Research Scientific Computing Center (NERSC). Cori log traces. https://docs.nersc.gov/systems/cori/.

[128] T. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Trans. Electron Devices*, 41(4):553–557, 1994.

[129] H. Okamura and T. Dohi. Comprehensive evaluation of aperiodic checkpointing and rejuvenation schemes in operational software system. *Journal of Systems and Software*, 83(9):1591–1604, 2010.

[130] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea. End-to-End I/O Portfolio for the Summit Supercomputing Ecosystem. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM, 2019.

[131] E. Oró, V. Depoorter, A. Garcia, and J. Salom. Energy efficiency and renewable energy integration in data centres. strategies and modelling review. *Renewable and Sustainable Energy Reviews*, 42:429–445, 2015.

[132] T. Patel, R. Garg, and D. Tiwari. GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems. In *Proc. 18th USENIX Conf. on File and Storage Technologies*, page 103–120. USENIX Association, 2020.

[133] T. Patel, Z. Liu, R. Kettimuthu, P. Rich, W. Allcock, and D. Tiwari. Job characteristics on large-scale systems: long-term analysis, quantification, and implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

[134] T. Pesch, S. Schröders, H. J. Allelein, and J. F. Hake. A new markov-chain-related statistical approach for modelling synthetic wind power time series. *New Journal of Physics*, 17(5), 2015.

[135] A. Radovanovic. Our data centers now work harder when the sun shines and wind blows, Apr. 2020. https://blog.google/inside-google/infrastructure/data-centers-work-harder-sun-shines-wind-blows.

[136] T. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, 2003.

[137] M. Scharbrodt, T. Schickinger, and A. Steger. A new average case analysis for completion time scheduling. *J. ACM*, 53(1):121–146, 2006.

[138] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of DSN*, pages 249–258, 2006.

[139] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[140] K. Schroiff, P. Gemsjaeger, and C. Bolik. Cascading failover of a data management application for shared disk file systems in loosely coupled node clusters, 2006. US Patent 6,990,606.

[141] A. Z. S. Shahul and O. Sinnen. Scheduling task graphs optimally with A*. *The Journal of Supercomputing*, 51:310–332, 2010.

[142] P. Sigdel, X. Yuan, and N. Tzeng. Realizing best checkpointing control in computing systems. *IEEE TPDS*, 32(2):315–329, 2021.

[143] L. Silva and J. Silva. Using two-level stable storage for efficient checkpointing. *IEE Proceedings - Software*, 145(6):198–202, 1998.

[144] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[145] S. Song, K. Hwang, and Y.-K. Kwok. Risk-resilient heuristics and genetic algorithms for security-assured grid job scheduling. *IEEE Transactions on Computers*, 55(6):703–719, 2006.

[146] J. Sonnek, A. Chandra, and J. Weissman. Adaptive reputation-based scheduling on unreliable distributed infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1551–1564, 2007.

[147] A. Souza and A. Steger. The expected competitive ratio for weighted completion time scheduling. In *STACS*, pages 620–631, 2004.

[148] G. N. Srinivasa Prasanna and B. R. Musicus. The optimal control approach to generalized multiprocessor scheduling. *Algorithmica*, 15(1):17–49, 1996.

[149] D. Stirzaker. *Elementary Probability*. Cambridge University Press, 2 edition, 2003.

[150] O. Subasi, G. Kestor, and S. Krishnamoorthy. Toward a general theory of optimal checkpoint placement. In *CLUSTER*, pages 464–474. IEEE, 2017.

[151] O. Subasi, T. Martsinkevich, F. Zyulkyarov, O. Unsal, J. Labarta, and F. Cappello. Unified fault-tolerance framework for hybrid task-parallel message-passing applications. *IJHPCA*, 32(5):641–657, 2018.

[152] Tal Rosenberg. The computer that will change everything. *Chicago Magazine*, Jan. 2023. https://www.chicagomag.com/chicago-magazine/february-2023/the-computer-that-will-change-everything/.

[153] Z. Tan, L. Du, D. Feng, and W. Zhou. EML: An I/O scheduling algorithm in large-scale-application environments. *Future Generation Computer Systems*, 78:1091–1100, 2018.

[154] P. Team. Pegasus workflow generator. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, 2014.

[155] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. IO-Cop: Managing concurrent accesses to shared parallel file system. In *43rd International Conference on Parallel Processing (ICPP) Workshops*, pages 52–60. IEEE, 2014.

[156] D. Tiwari, S. Gupta, and S. S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Int. Conf. on Dependable Systems and Networks*, pages 25–36. IEEE, 2014.

[157] Top500. Top 500 Supercomputer Sites, Nov. 2022. https://www.top500.org/lists/2022/11/.

[158] S. Toueg and O. Babaoğlu. On the optimum checkpoint selection problem. *SIAM J. Comput.*, 13(3), 1984.

[159] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA*, 1992.

[160] J. D. Ullman. Np-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[161] United Nations Framework Convention on Climate Change. Kyoto protocol, 1997. http://unfccc.int/kyoto_protocol/items/2830.php.

[162] United Nations Framework Convention on Climate Change. Paris climate change conference, 2015. http://unfccc.int/meetings/paris_nov_2015/meeting/8926.php.

[163] Q. Wang and K. H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2):281–294, 1992.

[164] D. J. White. A survey of applications of markov decision processes. *Journal of the operational research society*, 44(11):1073–1096, 1993.

[165] C.-M. Wu, R.-S. Chang, and H.-Y. Chan. A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147, 2014.

[166] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In *ScalA'11*, pages 25–28, 2011.

[167] F. Yang and A. A. Chien. ZCCloud: Exploring Wasted Green Power for High-Performance Computing. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1051–1060, 2016.

[168] F. Yang and A. A. Chien. Large-scale and extreme-scale computing with stranded green power: Opportunities and costs. *IEEE Transactions on Parallel and Distributed Systems*, 29(5), Dec. 2017.

[169] Y. Yang, X. Shi, W. Liu, H. Jin, Y. Hua, and Y. Jiang. DDL-QoS: a dynamic I/O scheduling strategy of QoS for HPC applications. *Concurrency and Computation: Practice and Experience*, 33(7), 2021.

[170] D. Ye, D. Z. Chen, and G. Zhang. Online scheduling of moldable parallel tasks. *J. of Scheduling*, 21(6):647–654, 2018.

[171] D. Ye, X. Han, and G. Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009.

[172] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema. Analysis and modeling of time-correlated failures in large-scale distributed systems. *Parallel and Distrib. Syst. Report Series*, 2010.

[173] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.

[174] F. Zanon Boito, G. Pallez, L. Teylo, and N. Vidal. IO-SETS: Simple and efficient approaches for I/O bandwidth management. https://hal.inria.fr/hal-03648225, May 2022.

[175] R. H. M. Zargar and M. H. Yaghmaee Moghaddam. Development of a markov-chain-based solar generation model for smart microgrid energy management system. *IEEE Transactions on Sustainable Energy*, 11(2):736–745, 2020.

[176] B. Zha and H. Shen. Adaptively Periodic I/O Scheduling for Concurrent HPC Applications. *Electronics*, 11(9), 2022.

[177] C. Zhang and A. A. Chien. Scheduling challenges for variable capacity resources. In D. Klusáček, W. Cirne, and G. P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 190–209, Cham, 2021. Springer International Publishing.

[178] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, 1998.

Chapter 9

Publications, Reports, and Submissions

9.1 Articles in International Refereed Conferences

[C1] Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms; A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, H. Sun; IEEE International Conference on Cluster Computing; 2020.

[C2] Multi-Resource List Scheduling of Moldable Parallel Jobs under Precedence Constraints; L. Perotin, H. Sun, P. Raghavan; International Conference on Parallel Processing-ICPP; 2021.

[C3] Online Scheduling of Moldable Task Graphs under Common Speedup Models; A. Benoit, L. Perotin, Y. Robert, Hongyang Sun; International Conference on Parallel Processing-ICPP; 2022 (Best Paper award).

9.2 Articles in International Refereed Journals

[J1] Resilient Scheduling of Moldable Parallel Jobs to Cope With Silent Errors; A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, H. Sun; IEEE Transactions on Computers; 2022.

[J2] Checkpointing Workflows à la Young/Daly Is Not Good Enough; A. Benoit, L. Perotin, Y. Robert, H. Sun; ACM Transactions on Parallel Computing; 2022.

9.3 Research Reports

[R1] Resilient Scheduling of Moldable Parallel Jobs to Cope with Silent Errors; A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, H. Sun; Inria Research Report RR-9340; January 2021; https://inria.hal.science/hal-02614215v2/document

[R2] Checkpointing Workflows à la Young/Daly Is Not Good Enough; A. Benoit, L. Perotin, Y. Robert, H. Sun; Inria Research Report RR-9413; June 2021; https://inria.hal.science/hal-03264047v1/document

[R3] Checkpointing strategies to protect parallel jobs from non-memoryless fail-stop errors; A. Benoit, L. Perotin, Y. Robert, F. Vivien; Inria Research Report RR-9465; March 2022; https://inria.hal.science/hal-03610883v2/document

[R4] Revisiting I/O bandwidth-sharing strategies for HPC applications; A. Benoit, T. Hérault, L. Perotin, Y. Robert, F. Vivien; Inria Research Report RR-9502; April 2023; https://inria.hal.science/hal-04038011/document

9.4 Submissions

[S1] Checkpointing strategies to tolerate non-memoryless failures on HPC platforms; A. Benoit, L. Perotin, Y. Robert, F. Vivien; ACM Transactions On Parallel Computing; Submitted on 27-Dec-2022 (Currently in revision process).

[S2] Multi-Resource Scheduling of Moldable Workflows; L. Perotin, S. Kandaswamy, H. Sun, Padma Raghavan; Journal of Parallel and Distributed Computing; Submitted on 04-Jan-2023.

[S3] Revisiting I/O bandwidth-sharing strategies for HPC applications; A. Benoit, T. Hérault, L. Perotin, Y. Robert, F. Vivien; ACM Transactions on Parallel Computing; Submitted on 06-Apr-2023.

[S4] Risk-Aware Scheduling Algorithms for Variable Capacity Resources; A. Benoit, A. Chien, L. Perotin, Y. Robert, R. Wijayawardana, C. Zhang; International Conference on Parallel Processing-ICPP; Submitted on 21-Apr-2023.

[S5] Improved Online Scheduling of Moldable Task Graphs under Common Speedup Models; L. Perotin, H. Sun; ACM Transactions on Parallel Computing; Submitted on 22-Apr-2023.

Appendices

Appendix A. Proof of Theorem 24

We start with two lemmas before proving Theorem 24:

Lemma 32. *Let $z, a, b \in \mathbb{R}^+$ and $n \in \mathbb{N}^*$ defined in the following domain:*

$$\begin{aligned} 1 &\leq z \\ 1 &\leq n \\ \left(\frac{z-1}{z}\right)^{\frac{1}{n}} &\leq p_0 < 1 \\ 0 &< p_z < 1 - p_0 \end{aligned}$$

Then the following equation holds:

$$(1 - (p_0 + p_z))\frac{z[(p_0 + p_z)^n - p_0^n]}{1 - (p_0 + p_z)^n} - p_z(z-1) \geq 0 \tag{1}$$

Proof. If we multiply by $1 - (p_0 + p_z)^n$ and develop, we see that this equation is equivalent to proving that the polynomial $P(z, p_0, p_z)$ is nonnegative over its domain, where

$$\begin{aligned} &P(z, p_0, p_z) = z(p_0 + p_z)^n + z(p_0 + p_z)p_0^n \\ &+ (z-1)p_z(p_0 + p_z)^n - zp_0^n - z(p_0 + p_z)^{n+1} - (z-1)p_z \end{aligned}$$

Consider fixed values of of p_0 and p_z with $0 < p_z < 1 - p_0$. The polynomial $Q(z) = P(z, p_0, p_z)$ s affine in z. The range of z is $1 \leq z \leq \frac{1}{1-p_0^n}$, so it is sufficient to prove that $Q(1) \geq 0$ and $Q(\frac{1}{1-p_0^n}) \geq 0$ to get the result.

We compute easily that $Q(1) = (1-p_0-p_z)((p_0+p_z)^n - p_0^n) \geq 0$. Now, $1 - \frac{1}{1-p_0^n} = \frac{p_0^n}{1-p_0^n}$ and

$$Q(\frac{1}{1-p_0^n}) = \frac{1}{1-p_0^n}\left[((p_0 + p_z)^n - p_0^n)(1-p_0) - (p_0 + p_z)^n(1-p_0^n)p_z\right]$$

Letting $R(p_z) = ((p_0 + p_z)^n - p_0^n)(1 - p_0) - (p_0 + p_z)^n(1 - p_0^n)p_z$, we need to show that $R(p_z) \geq 0$ for $0 \leq p_z \leq 1 - p_0$. But we have $R(0) = R(1 - p_0) = 0$, and differentiating, $R'(p_z) = (p_0 + p_z)^{n-1}S(p_z)$ where

$$S(p_z) = n(1 - p_0) - p_0(1 - p_0^n) - p_z(1 - p_0^n)(1 + p_0)$$

We see that $S(p_z)$ is affine, positive then negative over $\mathbb{R}$, hence $R(p_z)$ is strictly increasing and then strictly decreasing over $\mathbb{R}$. Given that $R(0) = R(1-p_0) = 0$, $R(p_z)$ is nonnegative for $0 \leq p_z \leq 1 - p_0$, which concludes the proof. □

Lemma 33. *Let $\epsilon \in (0, 1)$ and $(X_1, X_2, \ldots, X_n)$ be n independent random variables such that X_i can take only two values: 0 and $M_i > 0$. We assume that for all i, $\mathbb{E}(max(X_{i,1}, X_{i,2}, \ldots, X_{i,n})) = 1$, where all the $X_{i,j}$ follow the same law as X_i. We finally define X_0, a constant random variable always equal to 1. Then if for all $i > 0$, $M_i \geq \frac{2}{\epsilon}$, we have $\mathbb{E}(Y) = \mathbb{E}(max(X_0, X_1, X_2, \ldots, X_n)) \leq 2 + \epsilon$.*

Proof. We define for all $i > 0$, $p_i = \mathbb{P}\{X_i = 0\} = 1 - \mathbb{P}\{X_i = X_i\}$. From the condition $\forall i > 0, \mathbb{E}(max(X_{i,1}, X_{i,2}, \ldots, X_{i,n})) = 1$, we obtain the following relation between M_i and p_i:

$$\forall i > 0, \mathbb{E}(\max(X_{i,1}, X_{i,2}, \ldots, X_{i,n})) = M_i(1 - p_i^n) = 1,$$

from which we derive:

$$\forall i > 0, M_i = \frac{1}{1 - p_i^n}$$
$$\forall i > 0, p_i^n = 1 - \frac{1}{X_i} \geq 1 - \frac{\epsilon}{2}$$

We can then upper bound $\mathbb{E}(Y)$ to obtain:

$$\begin{aligned}
\mathbb{E}(Y) &= \mathbb{E}(\max(X_0, X_1, X_2, \ldots, X_n)) \\
&\leq \sum_{i=0}^{n} \mathbb{E}(X_i) \\
&\leq 1 + \sum_{i=1}^{n} (1 - p_i) M_i \\
&\leq 1 + \sum_{i=1}^{n} \frac{1 - p_i}{1 - p_i^n} \\
&= 1 + \sum_{i=1}^{n} \frac{1}{\sum_{j=0}^{n-1} p_i^j} \\
&\leq 1 + \sum_{i=1}^{n} \frac{1}{n p_i^n} \\
&\leq 1 + \frac{n}{n(1 - \frac{\epsilon}{2})} \\
&= 1 + 1 + \frac{\epsilon}{2 - \epsilon} \leq 2 + \epsilon.
\end{aligned}$$

□

We are now ready to prove Theorem 24.

Proof of Theorem 24. The sketch of the proof is as follows. Let $R_0 = \frac{\mathbb{E}(Y)}{\max_i(\mathbb{E}(Z_i))}$. We fix $\epsilon \in (0, 1)$ arbitrarily small. We apply a set of transformations to the x_i so that they eventually satisfy the conditions described in Lemma 33, while decreasing the ratio by a factor at most $(1 + \epsilon)^3$. To show this, we will use the equation of Lemma 32. This will prove that the ratio is less than $(1 + \epsilon)^3(2 + \epsilon)$ for all ϵ, thus not greater than 2.

Transformation 1: Bound X_i

We prove that we can bound the X_i in such a way that the ratio is increased by a factor at most $(1 + \epsilon)$. Let $i > 0$ and f be the probability density function of Z_i, thus $\mathbb{E}(Z_i) = \int_0^\infty x f(x)\, dx$. First consider $g(z) = \int_0^z x f(x)\, dx$. We know that $g(z)$ increases monotonically towards $\mathbb{E}(Z_i)$ thus there exists M_i such that $g(M_i) \geq \mathbb{E}(Z_i)\,(1 - \frac{\epsilon}{2})$. We define Z_i'' as follows:

$$Z_i'' = Z_i \text{ if } Z_i \leq M_i$$
$$Z_i'' = 0 \text{ otherwise}$$

Then $\mathbb{E}(Z_i'') = g(M_i) \geq \mathbb{E}(Z_i)\,(1 - \frac{\epsilon}{2})$. We now define X_i' in a similar manner:

$$X_i' = X_i \text{ if } X_i \leq M_i$$
$$X_i' = 0 \text{ otherwise}$$

Clearly, if we let $Z'_i = \max(X'_{i,1}, X'_{i,2}, \ldots, X'_{i,n})$, with all the $X'_{i,j}$ corresponding to the bounded $X_{i,j}$, we obtain that

$$Z'_i = Z_i \text{ if } Z_i \leq M_i$$
$$Z'_i \geq 0 \text{ otherwise}$$

Thus $\mathbb{E}(Z'_i) \geq \mathbb{E}(Z''_i) \geq \mathbb{E}(Z_i)\,(1 - \frac{\epsilon}{2})$.
We can apply this for all i, and replace the X_i by the X'_i. Clearly $\mathbb{E}(Y)$ will decrease and $\max_i(\mathbb{E}(Z_i))$ decreases by a factor at most $(1 - \frac{\epsilon}{2})$. Thus after the first transformation, the new ratio R_1 will verify:

$$R_0 \leq \frac{R_1}{1 - \frac{\epsilon}{2}} = R_1 \left(1 + \frac{\epsilon}{2 - \epsilon}\right) \leq R_1(1 + \epsilon)$$

From now on, we assume that all the X_i are bounded by M_i.

Transformation 2: Discretize X_i

We prove that we can transform the X_i so that they take only a finite number of values and so that the ratio is increased by a factor at most $(1 + \epsilon)$.

For all $i > 0$, we split the domain of X_i, $[0, M_i]$ into $N_i = \lceil \frac{M_i}{\epsilon \mathbb{E}(Y)} \rceil$ segments so that each segment is smaller than $\epsilon \mathbb{E}(Y)$, and if X_i is in a segment we replace it by the largest value of the segment. This will naturally increase $\mathbb{E}(Y)$ by a factor at most $(1 + \epsilon)$ and it will also increase $\max_i(\mathbb{E}(Z_i))$. Thus the ratio R_2 will verify $R_1 \leq (1 + \epsilon)R_2$. More formally, we define X'_i as the following:

$$X'_i = \left\lceil \frac{X_i N_i}{M_i} \right\rceil \frac{M_i}{N_i}$$

We apply this change for all i and define $Y' = \max(X'_1, \ldots, X'_n)$ as well as $Z'_i = \max(X'_{i,1}, X'_{i,2}, \ldots, X'_{i,n})$ and we have:

$$\begin{aligned} Y' - Y &\leq \max_i (X'_i - X_i) \\ &\leq \left(\left\lceil \frac{X_i N_i}{M_i} \right\rceil - \frac{X_i N_i}{M_i} \right) \frac{M_i}{N_i} \leq \frac{M_i}{N_i} \leq \epsilon \mathbb{E}(Y) \\ \mathbb{E}(Y') &\leq (1 + \epsilon)\mathbb{E}(Y) \\ \mathbb{E}(Z'_i) &\geq \mathbb{E}(Z_i) \end{aligned}$$

We can replace the X_i by the X'_i, and after transformation 2 we have

$$R_0 \leq R_1(1 + \epsilon) \leq R_2(1 + \epsilon)^2$$

From now on, we assume that all the X_i can take a finite number of values bounded by M_i.

Transformation 3: Add a high value in the domain of X_i

In addition to needing that all the X_i take a finite number of values, we also want that X_i may be extremely large, in order to end up with the conditions described in Lemma 33. More precisely, for any i we define $M'_i = \max\left(\frac{2(1+\epsilon)\mathbb{E}(Z_i)}{\epsilon}, M_i\right)$, $p_i = \frac{\epsilon^2}{2(1+\epsilon)n^2}$ and X'_i as follows:

$$X'_i = X_i \text{ with probability } 1 - p_i$$
$$X'_i = M'_i \text{ otherwise (e.g., if } X_i \text{ is within } p_i$$
$$\text{proportion of its highest values)}$$

We define Z'_i and Y' accordingly. Then:

$$\begin{aligned}
\forall i, \mathbb{E}(Z'_i) - \mathbb{E}(Z_i) &\leq np_i M'_i \leq \frac{\epsilon \mathbb{E}(Z_i)}{n} \\
\mathbb{E}(Y') - \mathbb{E}(Y) &\leq \sum_{i=1}^{n} p_i M'_i \leq \sum_{i=1}^{n} \frac{\epsilon \mathbb{E}(Z_i)}{n^2} \\
&\leq \sum_{i=1}^{n} \frac{\epsilon \mathbb{E}(X_i)}{n} \leq \sum_{i=1}^{n} \frac{\epsilon \mathbb{E}(Y)}{n} \leq \epsilon \mathbb{E}(Y) \\
\forall i, \mathbb{E}(Z_i) &\leq \mathbb{E}(Z'_i) \leq (1+\epsilon)\mathbb{E}(Z_i) \\
\mathbb{E}(Y) &\leq \mathbb{E}(Y') \leq (1+\epsilon)\mathbb{E}(Y) \\
\forall i, M'_i &\geq \frac{2\mathbb{E}(Z'_i)}{\epsilon}
\end{aligned}$$

We replace the X_i by the X'_i (with maximum value $M_i := M'_i$) and the new ratio R_3 verifies:

$$R_0 \leq R_2(1+\epsilon)^2 \leq R_3(1+\epsilon)^3$$

From now on, we assume that all the x_i can take a finite number of values, with a maximal value M_i larger than $\frac{2E(Z_i)}{\epsilon}$.

Transformation 4: Normalize $\mathbb{E}(Z_i)$
For all i, we alter the X_i such that all the $\mathbb{E}(Z_i)$ becomes equal to one. In practice we do the following:

$$X'_i = \frac{X_i}{\mathbb{E}(Z_i)}$$

As usual, if we define Z'_i accordingly to Z_i under a draw $(X_{i,1}, \ldots, X_{i,n})$, as well as Y' accordingly to Y from a draw of $(X_1, \ldots, X_n)$, we straightforwardly have:

$$\begin{aligned}
\forall i, \mathbb{E}(Z'_i) &= 1 \\
\frac{\mathbb{E}(Y')}{\mathbb{E}(Y)} &\geq \frac{1}{\max_i(\mathbb{E}(Z_i))} \\
\frac{\max_i(\mathbb{E}(Z'_i))}{\max_i(\mathbb{E}(Z_i))} &= \frac{1}{\max_i(\mathbb{E}(Z_i))} \\
M'_i &\geq \frac{M_i}{\mathbb{E}(Z_i)} \geq \frac{2}{\epsilon}
\end{aligned}$$

As before we can replace the X_i by the X'_i, and the new ratio increases, so we have:

$$R_0 \leq R_3(1+\epsilon)^3 \leq R_4(1+\epsilon)^3 = \mathbb{E}(Y)(1+\epsilon)^3$$

From now on, we assume that the X_i verify: $\mathbb{E}(Z_i) = 1$, X_i can only take a finite number of different values, whose highest is at least $\frac{2}{\epsilon}$. We are getting closer to the conditions of Lemma 33, we just need to add X_0 (easy) and to transform the X_i so that they can take only two values.

Transformation 5: Add $X_0 = 1$
We add the random variable X_0 which is always equal to one. We adapt $Y := \max(X_0, X_1, \ldots, X_n)$. Clearly Y can only increase while $\max_i(\mathbb{E}(Z_i))$ is still equal to one. Thus

$$R_0 \leq R_4(1+\epsilon)^3 \leq R_5(1+\epsilon)^3 = \mathbb{E}(Y)(1+\epsilon)^3$$

From this point, we apply a transformation that zero out the smaller possible value for X_i, reducing the number of possible values while $E(Z_i)$ remains unchanged even though M_i and $E(Y)$ increases. The first step will be zeroing out all the positive values smaller than one.

Transformation 6: Remove the minimal strictly positive possible value for a $X_i \neq X_0$, if this value is at most 1 (to be processed iteratively until all the X_i can only be equal to 0 or larger than 1)

This is easy to understand: if z is the minimal strictly positive value that X_i can reach with probability p_z, with $z \leq 1$, and if we want to transform X_i so that $p_z = 0$ while keeping $\mathbb{E}(Z_i) = 1$, we will need to increase the other values (or increase their probability). Either way, if $X_i = z \leq 1$ or if $X_i = 0$ it is strictly the same for Y because $Y \geq X_0 \geq 1$. Thus this transformation can only increase $\mathbb{E}(Y)$.

Let us choose i such that X_i can be in $(0, 1]$ and fix z as the minimal strictly positive value that X_i can reach. (if such i does not exist, we move on to the next step). We let $p_0 \triangleq \mathbb{P}\{X_i = 0\}$, $p_z \triangleq \mathbb{P}\{X_i = z\}$ and $p_{z^+} \triangleq \mathbb{P}\{X_i > z\}$. Similarly, we define $P_0 \triangleq \mathbb{P}\{Z_i = 0\}$, $P_z \triangleq \mathbb{P}\{Z_i = z\}$ and $P_{z^+} \triangleq \mathbb{P}\{Z_i > z\}$. We first compute all these values using only p_0 and p_z:

$$\begin{aligned} p_{z^+} &= 1 - p_0 - p_z \\ P_0 &= p_0^n \\ P_z &= (p_0 + p_z)^n - P_0 = (p_0 + p_z)^n - p_0^n \\ P_{z^+} &= 1 - (P_0 + P_z) = 1 - (p_0 + p_z)^n \end{aligned}$$

The idea of the transformation is the following: we zero out p_z ($p_0 := p_0 + p_z$ and $p_z := 0$) which makes $\mathbb{E}(Z_i)$ decrease by zP_z. To balance that, we increase all the other possible values (and not their probability) by $X = \frac{zP_z}{P_{z^+}}$ which will increase $\mathbb{E}(Z_i)$ by zP_z. Formally, we define our new variable X_i' as follows:

$$\begin{aligned} X_i' &= 0 \text{ if } X_i \leq z \\ X_i' &= X_i + \frac{zP_z}{P_{z^+}} \text{ otherwise} \end{aligned}$$

Consider a draw $(x_1, \ldots, x_n)$ following $(X_1, \ldots, X_n)$, and compare $Y = \max(x_0, \ldots, x_n)$ and $Y' = \max(x_0, x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_n)$. If $x_i \leq z < 1$ or if $x_i' \leq Y$, $Y = Y'$; otherwise $Y' > Y$. Thus $\mathbb{E}(Y') > \mathbb{E}(Y)$, $E(Z_i') = 1$, and the ratio increases as well as M_i.

When we may not apply transformation 6 again, each X_i can only be equal to 0 or larger than 1, with a maximum possible value greater than $\frac{2}{\epsilon}$, and we can move on to the last transformation before concluding the proof.

Transformation 7: If some X_i can take more than two different values, remove its smallest strictly positive value (to be processed iteratively until meeting the conditions of Lemma 33)

We take an X_i and its minimum strictly positive value $z \geq 1$; we apply the same transformation as in transformation 6 although the analysis differs, i.e.

$$\begin{aligned} X_i' &= 0 \text{ if } X_i \leq z \\ X_i' &= X_i + \frac{zP_z}{P_{z^+}} \text{ otherwise} \end{aligned}$$

Consider a draw $(x_1, \dots, x_n)$ following $(X_1, \dots, X_n)$, and compare $Y = \max(x_0, \dots, x_n)$ and $Y' = \max(x_0, x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_n)$. If $x_i \leq z < 1$ or if $x'_i \leq Y$, $Y = Y'$; otherwise $Y' > Y$. Thus $\mathbb{E}(Y') > \mathbb{E}(Y)$, $E(Z'_i) = 1$, and the ratio increases as well as M_i. Straightforwardly if $x_i \leq Y$ then $Y' \geq Y$; otherwise $x_i = \max_{j=0}^{n}(x_j)$. There are two cases:

- **Case 1:** $x_i = z$. This happens with a probability $p_z \prod_{i=0, i\neq j}^{n} \mathbb{P}\{x_j \leq z\}$ and in this case $Y' - Y = \max_{i=0,i\neq j}(x_j) - z \geq 1 - z$.
- **Case 2:** $x_i > z$. This happens with a probability $p_{z^+} \prod_{i=0, i\neq j}^{n} \mathbb{P}\{x_j \leq x_i | x_i > z\} \geq p_{z^+} \prod_{i=0, i\neq j}^{n} \mathbb{P}\{x_j \leq z\}$ and in this case $Y' - Y = \frac{zP_z}{P_{z^+}}$.

We are now able to bound $\mathbb{E}(Y' - Y)$ and show that it is nonnegative.

$$\mathbb{E}(Y' - Y) \geq \prod_{i=0, i\neq j}^{n} \mathbb{P}\{x_j \leq z\} \left(p_{z^+} \frac{zP_z}{P_{z^+}} - p_z(z-1) \right)$$

If $\prod_{i=0, i\neq j}^{n} \mathbb{P}\{x_j \leq z\} = 0$, we are done; otherwise:

$$\begin{aligned}
&\mathbb{E}(Y' - Y) \geq 0 \\
&\Leftrightarrow p_{z^+} \tfrac{zP_z}{P_{z^+}} - p_z(z-1) \geq 0 \\
&\Leftrightarrow (1 - (p_0 + p_z)) \tfrac{z[(p_0+p_z)^n - p_0^n]}{1-(p_0+p_z)^n} - p_z(z-1) \geq 0
\end{aligned}$$

We are now under the conditions described in Lemma 32 and can conclude. Indeed, the following conditions are obvious:

$$\begin{aligned}
n &\in \mathbb{N}^* \\
1 &\leq z \\
1 &\leq n \\
p_0 &< 1 \\
0 < p_z &< 1 - p_0
\end{aligned}$$

So we only need to show that $p_0 \geq (\frac{z-1}{z})^{\frac{1}{n}}$, i.e. $P_0 \geq \frac{z-1}{z}$. A quick study of $\mathbb{E}(Z_i)$ shows us what we need:

$$\begin{aligned}
&1 = \mathbb{E}(Z_i) = (1 - P_0)\mathbb{E}(Z_i | Z_i > 0) \geq (1 - P_0)z \text{ thus} \\
&P_0 \geq 1 - \frac{1}{z} \geq \frac{z-1}{z}
\end{aligned}$$

Applying Lemma 32, we finally show that $\mathbb{E}(Y') \geq \mathbb{E}(Y)$. We fix $x_i := x'_i$ and we have increased the ratio while decreasing the number of possible values for x_i, increasing M_i and keeping $\mathbb{E}(Z_i) = 1$. Once all the x_i can only take two possible values, 0 and $X_i \geq \frac{2}{\epsilon}$, we cannot apply transformation 7 any more, and are ready to conclude.

Conclusion of the proof of Theorem 24:
We are now exactly under the conditions of Lemma 33, Furthermore, transformations 6 and 7 increased the ratio, thus:

$$R_0 \leq (1 + \epsilon)^3 \mathbb{E}(Y) \leq (1 + \epsilon)^3 (2 + \epsilon)$$

Now suppose there exists a case such that $R_0 = 2 + \mu$ with $\mu > 0$. Applying the transformations with ϵ small enough (for example $\epsilon = min(\frac{1}{2}, \frac{\mu}{22})$) we reach a contradiction. We can finally conclude the proof and claim:

$$R_0 \leq 2$$

Tightness:
For any $\epsilon > 0$, it is possible to build an example such that $R_0 \geq 2 - \frac{1}{n} - \epsilon$. We provide a brief argument as follows. Consider n independent positive random variables, $X_1 = 1$ and for $i > 1$, $X_i = 0$ with probability p and x otherwise, such that $\mathbb{E}(Z_i) = 1$. As in Lemma 33 we have $x = \frac{1}{1-p^n}$. Then

$$\begin{aligned}\mathbb{E}(Y) &= \mathbb{P}\{Y = 1\} + X_i\mathbb{P}\{Y = x\} \\ &= p^{n-1} + \frac{1 - p^{n-1}}{1 - p^n} \\ &= 1 + p^{n-1} - \frac{1}{\sum_{i=0}^{n-1} p^i} \underset{p\to 1}{\longrightarrow} 2 - \frac{1}{n}\end{aligned}$$

This shows that we can build an example with a ratio arbitrarily close to $2 - \frac{1}{n}$. □

Appendix B. Detailed description of BestNextEvent

In this appendix, we detail Algorithm BestNextEvent, which was sketched in Section 7.4.3.

The aim of algorithm BestNextEvent is to maximize the minimum yield lexicographically at the next *predictable* event, that is, either at the end of the execution window or the first time one of the current communication requests is completed, whichever comes first. BestNextEvent is called each time an event occurs, either the completion of a communication request, the release of a new communication request, or the beginning of a new execution window. Then BestNextEvent defines a *constant* bandwidth allocation that will be applied up to the next event.

The algorithm itself is the combination of three algorithms. Algorithm BestNextEvent itself partitions the whole execution window in a set of what we call "simple intervals". Searching for an event that maximizes the minimum yield is relatively easy in a "simple interval" because the peculiar events which may change the nature of the optimal solution can only happen at the extremities of a simple interval. Algorithm BestNextEvent is described in Section 25. Once BestNextEvent has partitioned the whole execution window it searches, in each simple interval, and for each application that can define the next event in that interval, the solution maximizing the minimum yield by calling algorithm MinYieldInInterval (described in Section 9.4). Then, among all the solutions maximizing the minimum yield, BestNextEvent picks the one that maximizes the yield lexicographically by calling Algorithm LexicoMinYield (Section 20).

Before describing in detail algorithm MinYieldInInterval we describe its working principle.

Algorithm principle. Even if algorithm MinYieldInInterval looks complicated because of the many cases it has to deal with, its principle is rather simple. Let us consider a time $t+u$ and an application $\mathcal{A}_k$ defining an event at that time. Because application $\mathcal{A}_k$ defines an event at time $t+u$ its communication ends precisely at that time, it uses a bandwidth $\frac{\mathcal{V}_k}{u}$ and its yield is $y_k\left(t+u, \frac{\mathcal{V}_k}{u}\right)$. The remaining bandwidth is used to maximize the minimum yield achieved by the other applications. This minimum yield is defined either by the maximal bandwidth of an application ($\min_i y_i\left(t+u, b_i\right)$) or by the amount of remaining bandwidth. In the latter case, it turns out that all applications receiving some bandwidth achieve the same yield.

So far, we have described the algorithm principle for a given value of u. However, whatever the type of solution found (yield defined by the event-defining application, the maximal bandwidth of an application, etc.), the solution is valid in a neighbourhood of u. Therefore we study whether the minimum yield increases in this neighbourhood. Finally the neighbourhood itself is defined by the set of conditions defining the solution: which applications must receive bandwidths (which depends on the rankings and thus the intersection of the functions $\min_i y_i\left(t+u, 0\right)$), etc. In many cases we will have to determine the extent of this neighbourhood.

Preliminary remark. If the communication requests can not saturate the available bandwidth, that is, if $\sum_{i\in S(t)} b_i \leq B$ then, obviously, each application is alloted its maximum bandwidth and there is nothing to discuss. Otherwise, there is nothing we can do to optimize the yield of a computing application and the yield of an application is increasing while it computes. Hence, in the following, we only consider applications which have posted I/O operations.

Let t denote the date of the considered event. For each application $\mathcal{A}_i$ which has an I/O operation pending at time t (i.e., $i \in S(t)$), let $\mathcal{T}_i$ be the (minimum) time it would have required application $\mathcal{A}_i$ to progress as much as it did by time t *if it was the sole application running on the platform.* If $\mathcal{A}_i$ is the only application running on the platform them $\mathcal{T}_i = t - r_i$.

Maximizing the minimum yield in an interval

Let us consider a sub-interval $[t + u_{\min}, t + u_{\max}]$ of the window and the case where application $\mathcal{A}_k$ defines the next event. We will later loop over all applications to find the overall best solution in $[t + u_{\min}, t + u_{\max}]$. As stated in the introduction of this section, in order to simplify the study and the computations, we assume that the sub-interval $[t + u_{\min}, t + u_{\max}]$ is "simple", that is, that it does not contain any "peculiar" events, except maybe at its extremities ($t + u_{\min}$ and/or $t + u_{\max}$). We will define these "peculiar" events and show how to compute them during the algorithm walkthrough. Algorithm MinYieldInInterval (Algorithm 7) shows how to find the solution maximizing the minimum yield in the considered interval.

Algorithm MinYieldInInterval starts by studying the situation at time $t + u_{\min}$. That is, we consider the case where the next-event happens at time $t + u = t + u_{\min}$. Because $\mathcal{A}_k$ is the event-defining application, its communication completes at time $t + u_{\min}$ and it is allocated a bandwidth $\frac{\mathcal{V}_k}{u_{\min}}$. Hence, $\alpha_k = \frac{\mathcal{V}_k}{u_{\min} b_k}$ and the remaining bandwidth will be distributed among the other applications (Step 1).

The algorithm then computes an upper bound on the achievable minimum yield. Let us consider any application $\mathcal{A}_i$. The maximum bandwidth allocatable to $\mathcal{A}_i$ is b_i. Hence, the earliest time $\mathcal{A}_i$'s communication request could complete is at time $t + \frac{\mathcal{V}_i}{b_i}$. This time is a peculiar time and thus we add the set

$$\left\{ \frac{\mathcal{V}_i}{b_i} \mid i \in S(t) \right\}$$

to the set of peculiar times (Step 3 of Algorithm 8).

By hypothesis, if the next event happens at time $t + u$, no communication request can complete in the interval $(t, t + u)$. Therefore, because the yield of an application at time $t + u$ is a decreasing function of its allocated bandwidth, the maximum achievable yield for application $\mathcal{A}_i$ at time $t + u$ is

$$y_i^{\max}(t + u) = \begin{cases} y_i(t + u, b_i) & \text{if } u \leq \frac{\mathcal{V}_i}{b_i} \\ y_i\left(t + u, \frac{\mathcal{V}_i}{u}\right) & \text{otherwise.} \end{cases} \tag{2}$$

MinYieldInInterval starts by computing (at Step 2) an upper bound on the maximal minimum yield, that is, the minimum, over all communicating applications, of their maximum achievable yield, and of the yield of the event-defining application $\mathcal{A}_k$. We then check whether there is enough bandwidth overall to achieve this upper bound (Step 3).

The yield upper-bound is not achievable. We first consider the case where there is *not* enough bandwidth overall to achieve the upper bound. We want to compute the optimal yield y^{opt}. If no bandwidth is allocated to an application $\mathcal{A}_i$, it achieves a yield of $y_i(t + u_{\min}, 0)$ at time $t + u_{\min}$. Therefore, the only applications to which bandwidth is allocated are those such that $y^{opt} > y_i(t + u_{\min}, 0)$. At Step 4, MinYieldInInterval builds and sorts the set $\mathcal{Y}_0$ of the minimum yields achieved by the different applications (application $\mathcal{A}_k$ excepted) if they are not allocated any bandwidth. Let us denote by l the index of the element of $\mathcal{Y}_0$ such that $\mathcal{Y}_0[l] \leq y^{opt}$ and, either $y^{opt} < \mathcal{Y}_0[l + 1]$ or $l = |\mathcal{Y}_0|$

Algorithm 7: MinYieldInInterval($\mathcal{A}_k, u_{\min}, u_{\max}, \mathcal{S}$)

$RB \leftarrow B - \mathcal{V}_k/u_{\min}$ /* Remaining bandwith */
$y^{UB} \leftarrow \min\left\{\min_{i\in\mathcal{S}} y_i^{\max}(t+u_{\min}), y_k\left(t+u_{\min}, \frac{\mathcal{V}_k}{u_{\min}}\right)\right\}$
if $\sum_{i\in\mathcal{S}} \mathcal{BW}_i\left(t+u_{\min}, y^{UB}\right) > RB$ **then**
  $\mathcal{Y}_0 \leftarrow sort(\{y_i(u_{\min}, 0) \mid i \in \mathcal{S}\})$
  Find l such that $\sum_{i\in\mathcal{S}} \mathcal{BW}_i(u_{\min}, \mathcal{Y}_0[l]) \leq RB$ and $l = |\mathcal{Y}_0|$ or $\sum_{i\in\mathcal{S}} \mathcal{BW}_i(u_{\min}, \mathcal{Y}_0[l+1]) > RB$
  $\mathcal{I} \leftarrow \{i \in \mathcal{S} | y_i(u_{\min}, 0) \leq \mathcal{Y}_0[l]\}$
  $y_{\mathcal{I}}^{opt}(u) = \frac{uB - \mathcal{V}_k + \sum_{i\in\mathcal{I}} b_i\mathcal{T}_i}{\sum_{i\in\mathcal{I}} b_i(t+u-r_i)}$
  if $y_{\mathcal{I}}^{opt}(u)$ *is a non-increasing function* **then return** $(y_{\mathcal{I}}^{opt}(u_{\min}), u_{\min})$
  If $l < \mathcal{Y}_0$ let $\mathcal{A}_j$ be such that $y_j(u_{\min}, 0) = \mathcal{Y}_0[l+1]$ and $y_j(u_{\max}, 0)$ is minimal
  Let $u_{\text{intersect}}$ be the first intersection (if it exists) in $[u_{\min}, u_{\max}]$ of $y_{\mathcal{I}}^{opt}(u)$ with either $y_k\left(t+u, \frac{\mathcal{V}_k}{u}\right)$ or $y_j(t+u, 0)$ (if the later exists) or with $y_m(t+u, b_m)$ for some $m \in \mathcal{I}$
  if $u_{intersect}$ *does not exist* **then return** $(y_{\mathcal{I}}^{opt}(u_{\max}), u_{\max})$
  else if $y_{\mathcal{I}}^{opt}(u_{intersect}) = y_k\left(t+u_{intersect}, \frac{\mathcal{V}_k}{u_{intersect}}\right)$ **then return** $\left(y_k\left(t+u_{\text{intersect}}, \frac{\mathcal{V}_k}{u_{\text{intersect}}}\right), u_{\text{intersect}}\right)$
  else return MinYieldInInterval($\mathcal{A}_k, u_{\text{intersect}}, u_{\max}, \mathcal{S}$)
if $y^{UB} = y_k\left(t+u_{\min}, \frac{\mathcal{V}_k}{u_{\min}}\right)$ **then**
  return $\left(y_k\left(t+u_{\min}, \frac{\mathcal{V}_k}{u_{\min}}\right), u_{\min}\right)$
else
  Let $\mathcal{A}_j$ be such that $y_j^{\max}(t+u_{\min}) = y^{UB}$ and $y_j^{\max}(t+u_{\max})$ is minimal
  if $\mathcal{BW}_j\left(u_{\min}, y_j^{max}(t+u_{\min})\right) = \frac{\mathcal{V}_j}{u_{\min}}$ **then return** $(y_j^{\max}(t+u_{\min}), u_{\min})$
  $\mathcal{I} \leftarrow \{i \in \mathcal{S} \mid y_i(u_{\min}, 0) \leq y^{UB}\}$
  $y_{\mathcal{I}}^{opt}(u) = \frac{uB - \mathcal{V}_k + \sum_{i\in\mathcal{I}} b_i\mathcal{T}_i}{\sum_{i\in\mathcal{I}} b_i(t+u-r_i)}$
  If $\mathcal{I} \neq \mathcal{S}$ then let $\mathcal{A}_l$ be an application in $\mathcal{S} \setminus \mathcal{I}$ such that $y_l\left(\frac{u_{\min}+u_{\max}}{2}, 0\right)$ is minimal
  Let $u_{\text{intersect}}$ be the first intersection (if it exists) in $[u_{\min}, u_{\max}]$ of $y_j^{\max}(t+u)$ with $y_{\mathcal{I}}^{opt}(u)$, $y_k\left(t+u, \frac{\mathcal{V}_k}{u}\right)$, or $y_l(t+u, 0)$
  if $u_{intersect}$ *does not exist* **then return** $y_j^{\max}(t+u_{\max})$
  if $y_j^{max}(t+u_{intersect}) = y_k\left(t+u_{intersect}, \frac{\mathcal{V}_k}{u_{intersect}}\right)$ **then return** $y_k\left(t+u_{\text{intersect}}, \frac{\mathcal{V}_k}{u_{\text{intersect}}}\right)$
  else return MinYieldInInterval($\mathcal{A}_k, u_{\text{intersect}}, u_{\max}, \mathcal{S}$)

(i.e., $\mathcal{Y}_0[l]$ is the largest element in $\mathcal{Y}_0$ smaller than or equal to y^{opt}). l is computed at Step 5 either in linear time through an exhaustive search or, more cleverly, in logarithmic time through a binary search. Then, bandwidth should be allocated to all applications if $l = |\mathcal{Y}_0|$ and, otherwise, bandwidth should only be allocated to applications such that $y_i\left(t + u_{\min}, 0\right) \leq \mathcal{Y}_0[l]$. The yield of each of these applications is then equal to y^{opt}. Indeed, by definition of y^{opt} the yield of an application cannot be smaller. Because we are in the case where the upper-bound on the yield is not achievable, each application (if allocated enough bandwidth) can achieve a yield strictly larger than y^{opt} at time $t + u_{\min}$ without violating its bandwidth limit b_i. Then, if the yield of one application receiving some bandwidth was strictly larger than y^{opt}, some of its bandwidth could be distributed to the other applications to increase the value of y^{opt} which would contradict the optimality of y^{opt}.

Finally, the total bandwidth should be distributed among the applications. Hence, the constraints on the distribution of bandwidth are:

$$\forall i \in \mathcal{I}, y_{\mathcal{I}}^{opt}(u) = \frac{\mathcal{T}_i + \alpha_i u}{t + u - r_i} \qquad \text{and} \qquad \sum_{i \in \mathcal{I}} \alpha_i b_i = B - \frac{\mathcal{V}_k}{u}$$

where $\mathcal{I}$ is the set of the communicating applications requiring bandwidth, as defined at Step 6. We wrote the constraints for an undefined variable u rather than just for the value $u_{\min}$ for which we know there are true. This is because we are next going to study the evolution of the defined solution in a neighbourhood of $t + u_{\min}$.

From the first equation we obtain:

$$(t + u - r_i) y_{\mathcal{I}}^{opt}(u) = \mathcal{T}_i + \alpha_i u \qquad \Leftrightarrow \qquad \frac{1}{u}\left((t + u - r_i) y_{\mathcal{I}}^{opt}(u) - \mathcal{T}_i\right) = \alpha_i.$$

Then the second equation can be rewritten:

$$\sum_{i \in \mathcal{I}} \frac{b_i}{u}\left((t + u - r_i) y_{\mathcal{I}}^{opt}(u) - \mathcal{T}_i\right) = RB \quad \Leftrightarrow \quad y_{\mathcal{I}}^{opt}(u) = \frac{uRB + \sum_{i \in \mathcal{I}} b_i \mathcal{T}_i}{\sum_{i \in \mathcal{I}} b_i (t + u - r_i)}$$

Hence:

$$y_{\mathcal{I}}^{opt}(u) = \frac{uB - \mathcal{V}_k + \sum_{i \in \mathcal{I}} b_i \mathcal{T}_i}{\sum_{i \in \mathcal{I}} b_i (t + u - r_i)}. \tag{3}$$

We then study the variations of $y_{\mathcal{I}}^{opt}(u)$:

$$\begin{aligned} y_{\mathcal{I}}^{opt}(u) &= \frac{uB - \mathcal{V}_j + \sum_{i \in \mathcal{I}} b_i \mathcal{T}_i}{\sum_{i \in \mathcal{I}} b_i (t + u - r_i)} \\ &= \frac{B}{\sum_{i \in \mathcal{I}} b_i} + \frac{1}{\sum_{i \in \mathcal{I}} b_i} \frac{\left(\sum_{i \in \mathcal{I}} b_i\right)\left(\sum_{i \in \mathcal{I}} b_i \mathcal{T}_i\right) - \left(\sum_{i \in \mathcal{I}} b_i (B(t - r_i) + \mathcal{V}_j)\right)}{\sum_{i \in \mathcal{I}} b_i (t + u - r_i)}. \end{aligned}$$

Therefore, the yield is increasing with u on the considered interval if and only if the expression

$$\left(\sum_{i \in \mathcal{I}} b_i\right)\left(\sum_{i \in \mathcal{I}} b_i \mathcal{T}_i\right) - \left(\sum_{i \in \mathcal{I}} b_i (B(t - r_i) + \mathcal{V}_j)\right) \tag{4}$$

is negative. Hence, if this expression is non-negative, the yield is non-increasing over the interval and the optimum is found for $u = u_{\min}$ (Step 8).

If the yield is increasing, things are more complicated. Equation (3) defines the optimum yield for a given set $\mathcal{I}$ of applications among which the remaining bandwidth should be distributed. In turn, the definition of $\mathcal{I}$ depends on the considered time $t + u$ and on the

amount of remaining bandwidth which is also a function of u. Let us assume that the interval $(t + u_{\min}, t + u_{\max})$ is such that no two curves $u \mapsto y_i(t + u, 0)$ intersects in this interval (except if the two curves are identical). Therefore, the intersection of two curves $u \mapsto y_i(t + u, 0)$ defines a "peculiar event" and the set of these intersections is computed at Step 9 of Algorithm 8. Then, if $l = |\mathcal{Y}_0|$, there is no curve $u \mapsto y_i(t + u, 0)$ above the curve $u \mapsto y_{\mathcal{I}}^{opt}(u)$. Otherwise, let application $\mathcal{A}_j$ be an application whose curve is the first one above the curve $u \mapsto y_{\mathcal{I}}^{opt}(u)$. In practice, let application $\mathcal{A}_j$ be an application such that $y_j(t + u_{\min}, 0) = \mathcal{Y}_0[l + 1]$ and such that $y_j(t + u_{\max}, 0)$ is minimal (Step 9). When the yield is increasing, it may intersects the curve $y_j(t + u, 0)$, changing the definition of the set $\mathcal{I}$ of applications requiring bandwidth. If this happens at a date $t + u_{\text{intersect}}$, then we know that the maximal min yield on the interval $[u_{\min}, u_{\text{intersect}}]$ is achieved for $u_{\text{intersect}}$. Then, we recursively call Algorithm MinYieldInInterval on the interval $[u_{\text{intersect}}, u_{\max}]$ (Step 13).

Another potential problem when the yield is increasing is that the (minimum) yield of the applications receiving bandwidth, $y_{\mathcal{I}}^{opt}(u)$, becomes equal to the yield of the event-defining applications, $\mathcal{A}_k$, at some date $t + u_{\text{intersect}}$. Then, we know that the maximal min yield on the interval $[u_{\min}, u_{\text{intersect}}]$ is achieved for $u_{\text{intersect}}$. Furthermore, because the function $y_k\left(t + u, \frac{\mathcal{V}_k}{u}\right)$ is non-increasing, the minimum yield is non increasing on the interval $[u_{\text{intersect}}, u_{\max}]$. Hence, the maximum minimum yield is obtained at time $t + u_{\text{intersect}}$ (Step 12).

The last problem that may happen, when u is increasing, is that the bandwidth allocated to an application $\mathcal{A}_m$ may increase so much that it reaches its limit b_m, at some date $t + u_{\text{intersect}}$. At that point, the nature of the solution changes. We know that the maximal min yield on the interval $[u_{\min}, u_{\text{intersect}}]$ is achieved for $u_{\text{intersect}}$ and, once again, we recursively call Algorithm MinYieldInInterval on the interval $[u_{\text{intersect}}, u_{\max}]$ (Step 13).

All the "problems" we just highlighted are defined by the equality of two yield functions. All these yield functions are of the form $\frac{\alpha u+\beta}{\gamma u+\delta}$. Therefore, looking for the equality of two such functions requires to solve a second degree polynomial and to see whether it has roots in the interval $[u_{\min}, u_{\max}]$ and, if there are two of them, which one is the smallest. Therefore, the algorithm can easily check, at Step 10, whether any of these potential problems occurs and if this is the case, which one happens the earliest. If no problem occurs, $y_{\mathcal{I}}^{opt}(u)$ defines a valid, increasing, solution throughout the interval and the best solution is found at time $t + u_{\max}$ (Step 11). Otherwise, the algorithm applies the relevant case, as defined above.

The yield upper-bound is achievable. We now consider the case where there is enough bandwidth overall to achieve the upper bound. We first check whether the application defining the upper-bound is also the event-defining application $\mathcal{A}_k$ (Step 14). If this is the case, because the yield of the event-defining application is a non-increasing function, $y_k\left(t + u, \frac{\mathcal{V}_k}{u}\right)$, the optimal solution on the interval $[t + u_{\min}, t + u_{\max}]$ is obtained in $u_{\min}$ (Step 15).

Otherwise, we identify (Step 17) an application, say $\mathcal{A}_j$, which defines the upper bound on the minimum yield at time $t + u_{\min}$. If there are several candidates we pick one which also defines this bound at the end of the interval; this is possible because we assume the interval includes no *peculiar* events. Therefore we add to the set of peculiar events the times at which two curves $u \mapsto y_i^{\max}(t + u)$ intersect (Steps 5, 6, and 7 of Algorithm 8). If there are still several candidates we pick one arbitrarily as they all have the same maximum yield throughout the interval.

The yield achieved by application $\mathcal{A}_j$, namely $y_j^{\max}(t+u)$, is an increasing function of u. This yield defines the optimal minimum yield as long as two conditions hold: 1) it is not greater than the yield $y_k\left(t+u, \frac{\mathcal{V}_k}{u}\right)$ of application $\mathcal{A}_k$; 2) the bandwidth required for all applications to achieve a yield at least equal to $y_j^{\max}(t+u)$ does not exceed the total available bandwidth. Algorithm MinYieldInInterval first identifies (at Step 19) to which applications bandwidth should be allocated for all applications to have a yield of at least $y_j^{\max}(t+u)$. Then it computes the maximal yield $y_{\mathcal{I}}^{opt}(u)$ achieved when the total remaining bandwidth is distributed among these applications. Note that this equation is only meaningful when one must distribute at least the total remaining bandwidth to achieve a yield at least equal to $y_j^{\max}(t+u)$. Finally, the algorithm computes the latest time, in the interval $[t+u_{\min}, t+u_{\max}]$, at which the two above conditions hold (Step 22): it computes the earliest time $t+u_{\text{intersect}}$ in $[t+u_{\min}, t+u_{\max}]$ (if it exists) when the curves $y_j^{\max}(t+u)$ and $y_k\left(t+u, \frac{\mathcal{V}_k}{u}\right)$ intersect or when the curves $y_j^{\max}(t+u)$ and $y_{\mathcal{I}}^{opt}(u)$ intersect.[1]

If $u_{\text{intersect}}$ does not exist, the optimal solution is $y_j^{\max}(t+u_{\max})$ (Step 23). Otherwise, the best solution in $[t+u_{\min}, t+u_{\text{intersect}}]$ is reached at time $t+u_{\text{intersect}}$ at which time the nature of the solution changes. If $y_j^{\max}(t+u_{\text{intersect}}) = y_k\left(t+u_{\text{intersect}}, \frac{\mathcal{V}_k}{u_{\text{intersect}}}\right)$ then, like previously, the best solution in $[t+u_{\min}, t+u_{\max}]$ is reached in $t+u_{\text{intersect}}$ (Step 24). Otherwise, Algorithm MinYieldInInterval is recursively called on the interval $[u_{\text{intersect}}, u_{\max}]$ (Step 25).

Maximizing the minimum yield in the whole execution window

BestNextEvent starts by partitioning the whole (remaining) execution window $[t, T_{end}]$ based on the peculiar events identified in the previous section. This is done at Step 1 and at Steps 5 through 10.

We compute at Step 1 the earliest time each communication can complete. If none of these dates happens before the end of the window, the next event happens at the end of the window (Step 2). If there is enough total bandwidth for all communications to be allocated their maximum bandwidth, the next event happens the first time a communication can complete (Step 3). Otherwise, BestNextEvent scans one by one the intervals defined by the peculiar events. For each of these intervals, and for each application whose communication can complete in the studied interval, it calls MinYieldInInterval to find the best possible solution. The best solution, among all the identified candidates, is eventually selected at Step 20.

The test at Step 18 enables us to implement an optimization: once a candidate solution is identified where the event-defining application is also the application with minimum yield then, because the yield $y_k\left(t, \frac{\mathcal{V}_k}{t}\right)$ is a decreasing function, and it is the highest yield application $\mathcal{A}_k$ can achieve at time t, we know that no better solution can be found for larger values of t and the search can stop.

Maximizing the minimum yield lexicographically at time $t+u$

LexicoMinYield (see Algorithm 9) is an algorithm which builds a bandwidth allocation that lexicographically maximizes the minimum yield at the date $t+u$ (if no event occurs in

[1] It may happen (especially after a recursive call), that $y_j^{\max}(t+u_{\min}) = y_{\mathcal{I}}^{opt}(u_{\min})$. In such a case, the algorithm determines which of the two functions $y_j^{\max}(t+u)$ and $y_{\mathcal{I}}^{opt}(u)$ achieves the smallest yield on $[t+u, t+u+\epsilon]$, that is, when u is infinitesimally greater than $u_{\min}$. If it is $y_j^{\max}(t+u)$, it is used instead of $y_{\mathcal{I}}^{opt}(u)$ to compute $u_{\text{intersect}}$.

Algorithm 8: BestNextEvent(t, T_{end})

$\mathcal{S} \leftarrow \left\{ \frac{\mathcal{V}_i}{b_i} \mid {}_{i \in S(t)} \right\} \cap [0,\, T_{end} - t]$
if $\mathcal{S} = \emptyset$ **then return** LexicoMinYield(t, $T_{end}, \emptyset$)
if $\sum_{i \in S(t)} b_i \leq B$ **then return** LexicoMinYield($t, \min \mathcal{S}, \emptyset$)
$\mathcal{D} \leftarrow \{(h, \emptyset)\}$
JointEvents $\leftarrow$
$\left\{ u \,\middle|\, \exists i,j \in S(t),\, T_{end} - t \geq u \geq \frac{\mathcal{V}_i}{b_i} \geq \frac{\mathcal{V}_j}{b_j}, y_i(u, \frac{\mathcal{V}_i}{u}) = y_j(u, \frac{\mathcal{V}_j}{u}), \mathcal{T}_i + \frac{\mathcal{V}_i}{bi} \neq \mathcal{T}_j + \frac{\mathcal{V}_j}{bj} \right\}$
$\mathcal{M}_1 \leftarrow$
$\left\{ u \,\middle|\, 0 \leq u \leq T_{end} - t, i, j \in S(t), i \neq j, y_i\left(u, b_i\right) = y_j\left(u, \frac{\mathcal{V}_j}{u}\right), u \leq \frac{\mathcal{V}_i}{b_i}, u \geq \frac{\mathcal{V}_j}{b_j} \right\}$
$\mathcal{M}_2 \leftarrow$
$\left\{ u \,\middle|\, 0 \leq u \leq T_{end} - t, i, j \in S(t), i \neq j, y_i\left(u, b_i\right) = y_i\left(u, b_j\right), u \leq \frac{\mathcal{V}_i}{b_i}, u \leq \frac{\mathcal{V}_j}{b_j} \right\}$
$\mathcal{X} \leftarrow$
$\left\{ u \,\middle|\, 0 \leq u \leq T_{end} - t, i, j \in S(t), i \neq j, T_{end} - t \geq u \geq \frac{\mathcal{V}_j}{b_j}, y_i\left(u, 0\right) = y_j\left(u, \frac{\mathcal{V}_j}{u}\right) \right\}$
$\mathcal{Z} \leftarrow \{u \,|0 \leq u \leq T_{end} - t, i, j \in S(t), i \neq j, 0 \leq u \leq T_{end} - t, y_i\left(u, 0\right) = y_j\left(u, 0\right)\}$
$\mathcal{U} \leftarrow sort(JointEvents \cup \mathcal{S} \cup \mathcal{X} \cup \mathcal{M}_1 \cup \mathcal{M}_2 \cup \{T_{end} - t\})$
DominantSolutionNotFound $\leftarrow$ *true*
$i \leftarrow 1$
while $i \leq |\mathcal{U}| - 1$ *and DominantSolutionNotFound* **do**
  for $k \in S(t)$ **do**
    if $u_i \geq \frac{\mathcal{V}_k}{b_k}$ **then**
      $(y, u) \leftarrow$ MinYieldInInterval($k, u_i, u_{i+1}, S(t) \setminus \{k\}$)
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{u, k\}$
      if $y = y_k\left(u, \frac{\mathcal{V}_k}{u}\right)$ **then** *DominantSolutionNotFound* $\leftarrow$ *false*
  $i \leftarrow i + 1$
return $\max_{(u,k) \in \mathcal{D}}$ LexicoMinYield(t, u, k)

the interval $(t, t+u)$) and an event defined by application $\mathcal{A}_k$ happens at time $t+u$. The algorithm principle is straightforward. It starts, if an event-defining application $\mathcal{A}_k$ is designed, to allocate it the required bandwidth (Steps 3 through 5). Then, LexicoMinYield calls, at Step 10, MinYield to obtain a bandwidth allocation maximizing the minimum yield. By definition, the yield of an optimal solution cannot be increased. The yield of a solution cannot be increased because achieving such a yield requires either to saturate the total available bandwidth, or to allocate one application its maximum allocatable bandwidth, or for a communication request to end at time $t+u$. In the first case, the yield of each communicating application is equal to the maximum minimum yield and an optimal solution has been built (Steps 12 and 14). For the other cases (which are not excluding), we fix the bandwidth of the applications whose communication ends at time $t+u$ and those whose allocated bandwidth is equal to the maximum allocatable one (Step 18). We compute the total remaining bandwidth (Step 19) and the minimum yield maximization is redone on the remaining applications (defined at Step 20) with the remaining bandwidth.

Algorithm LexicoMinYield has complexity of $O(m^2 \log(m))$, because of the complexity of MinYield and because the while loop is executed at most m times.

Algorithm MinYield is pretty straightforward. It finds the minimum yield that can be achieved among the set of applications whose bandwidth has not already been fixed, knowing the amount of bandwidth that remains to be allocated.

Algorithm 9: LexicoMinYield(t, u, k)

```
1  ∀i ∈ S(t) b_i ← 0                          /* Current bandwidth allocation */
2  if k ≠ ∅ then
3      b_k ← V_k / u
4      RB ← B − b_k                                  /* Remaining bandwith */
5      F ← S(t) \ {k}   /* Applications whose bandwith allotment is not yet
        fixed */
6  else
7      RB ← B                                        /* Remaining bandwith */
8      F ← S(t)         /* Applications whose bandwith allotment is not yet
        fixed */
9  while F ≠ ∅ and RB > 0 do
10     y ← MinYield(t, u, RB, F)
11     if Σ_{i∈F} BW_i(u, y) = RB then /* The whole remaining bandwidth is
        used                                                          */
12         ∀i ∈ F, b_i ← BW_i(u, y)
13         F ← ∅
14         RB ← 0
15     else
16         for i ∈ F do
17             if BW_i(u, y) = b_i or BW_i(u, y) u = V_i then
                   /* The application bandwidth cannot be increased */
18                 b_i ← BW_i(u, y)
19                 RB ← RB − b_i
20                 F ← F \ {i}
21 if RB > 0 then Error
```

Algorithm 10: MinYield$(t, u, B, \mathcal{S})$

$y^{\max} \leftarrow \min_{i \in \mathcal{S}} y_i^{\max}(u)$
if $\sum_{i \in \mathcal{S}} \mathcal{BW}_i(u, y^{\max}) \leq B$ **then return** $y^{\max}$
$\mathcal{Y}_0 \leftarrow sort(\{y_i(u, 0) \mid i \in \mathcal{S}\})$
Find l such that $\sum_{i \in \mathcal{S}} \mathcal{BW}_i(u_{\min}, \mathcal{Y}_0[l]) \leq B$ and $l = |\mathcal{Y}_0|$ or $\sum_{i \in \mathcal{S}} \mathcal{BW}_i(u_{\min}, \mathcal{Y}_0[l+1]) > RB$
$\mathcal{I} \leftarrow \{i \mid i \in \mathcal{S}, y_i(u, 0) \leq \mathcal{Y}_0[l]\}$
return $\dfrac{uB + \sum_{i \in \mathcal{I}} \mathcal{T}_i b_i}{\sum_{i \in \mathcal{I}} (t + u - r_i) b_i}$