Fault-tolerant algorithms for iterative applications and batch schedulers

Yishu Du


Numéro National de Thèse : 2022ENSL0045

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 06/12/2022, par :

Yishu DU

Fault-tolerant algorithms for iterative applications and batch schedulers

Algorithmes tolérants aux pannes pour les applications itératives et les ordonnanceurs

Devant le jury composé de :

Luc	GIRAUD	Directeur de Recherche Inria Bordeaux	Rapporteur
Marc	CASAS	Professeur, Barcelona Supercomputing Center	Rapporteur
Fanny	DUFOSSÉ	Chargée de Recherche, Inria Grenoble	Examinatrice
Francieli	ZANON BOITO	Maitresse de conférences, Université de Bordeaux	Examinatrice
Yves	ROBERT	Professeur des universités, ENS de Lyon	Directeur de thèse
Loris	MARCHAL	Chargé de Recherche - HDR, CNRS, LIP Lyon	Co-encadrant de thèse

Contents

Introduction

Due to the rapid development of information technology, computers are widely used in all fields of modern science and technology. Scientific computing has always been an important field of computer application. With the increase in computer memory and the speed of calculation, the size of problems to be solved is also increasing. In recent years, the rapid development of high-performance computers, also known as supercomputers, has brought many new challenges. In February 2014, ten challenges were identified to achieve the Exascale system by the Advanced Scientific Computing Advisory Committee, one of which is resilience, also known as fault-tolerance [70].

Deploying scientific applications at a large scale requires fault-tolerance mechanisms, which can mitigate the impact of errors and ensure a correct and uninterrupted execution of the application [19, 20]. But from a fault-tolerance standpoint, scale is the enemy. State-of-the-art supercomputers such as Frontier, Fugaku, or LUMI (respectively ranked 1st, 2nd, and 3rd in the TOP500 ranking [104]) are now embedding millions of cores (with a peak at 10.6M for Sunway TaihuLight (6th)). These large computing systems are frequently confronted with failures, also called fail-stop errors (such as hardware failures or crashes). Even if each of their cores has a very low probability of failure, the failure probability of the whole system is much higher. More precisely, assume that the *Mean Time Between Failure* (MTBF) of each computing resource is around ten years, meaning that such a resource should experience an error only every ten years on average, which explains why computing resources are individually very reliable. When running a simulation code on 100,000 of these resources in parallel, the MTBF is reduced to only 50 minutes [55]: one node of the computing platform crashes every 50 minutes on average. With one million such resources, the MTBF gets as small as five minutes, while codes deployed on such extreme-scale platforms usually last for hours or days. As the demand for computing power increases, failures cannot be ignored anymore, and fault-tolerance mechanisms must be deployed.

The classical way of dealing with failures in the extreme-scale computing systems consists of Checkpoint/Rollback mechanisms. A *checkpoint* of the application is taken periodically; that is, the state of the application (usually the whole content of its memory) is written onto reliable storage. Whenever one of the computing resources experiences a failure, the application pauses and restarts from the last valid checkpoint. Several studies have focused on the crucial question of the optimal checkpointing period, defined as the time between two consecutive checkpoints. On the one hand, if checkpoints are taken too often, time is wasted in costly I/O operations. On the other hand, if checkpoints are too infrequent, time will be wasted in recomputing large portions of the application after each failure. Interestingly, reliability was already a question in the early days of computing: in the 70s, Young proposed a first-order approximation of the optimal time between two checkpoints that minimizes the expected duration of the whole computation [110]. Young's approximation has then been refined by Daly thirty years later [29]. Young [110] and Daly [29] derived the well-known Young/Daly formula $\mathcal{P}_{YD} = \sqrt{2\mu_f C}$ for the optimal checkpointing period, where μ_f is the platform MTBF, and C is the checkpointing duration. Assuming unit speed, the time $\mathcal{P}_{YD}$ elapsed between two checkpoints is also the amount of work executed during

each period. The Young/Daly formula applies to applications where a checkpoint can be taken anytime during the computation. Divisible-load applications [13, 85] are an example of such applications. Also, the Young/Daly formula assumes that the time needed to take a checkpoint is constant (which corresponds to a constant size of the data to save).

This thesis makes several contributions to this important field of resilience for scientific computations. As described below, some of these contributions concern iterative applications that consist of a series of parallel tasks, while other contributions apply to batch schedulers which are vital modules of high-performance computers.

In addition to divisible-load applications, many scientific applications exhibit a more complicated behavior. In Chapter 2, we focus on *iterative applications* which we define as applications that are decomposed into computational *iterations*, where one can checkpoint only at the end of an iteration. Indeed, for iterative applications, checkpointing is efficient, let alone possible, only at the end of an iteration, because the volume of data to checkpoint is dramatically reduced at that point. A wide range of applications fits in this framework. Iterative solvers for sparse linear algebra systems are a representative example [83, 88]. Moreover, the time of each iteration depends upon several parameters (sparsity pattern of some vectors, communication contention, system jitter) and can vary significantly from one iteration to another. This phenomenon is amplified in randomized iterative methods [46] where random vectors are generated as the application progresses. Another class of applications that are naturally decomposed into iterations of variable length are Bulk Synchronous Parallel (BSP) applications [44, 57] where one checkpoints at the end of each join operation. A typical example of a BSP sequence of fork-join operations is the n-body computation [15]. Due to the simplicity of the programming model, many BSP applications are deployed at scale [14].

Scientific workflows account for a large fraction of the complex applications that are deployed on supercomputers, and they cover a wide range of domains, such as weather prediction, climate modeling, astronomy, and bioinformatics [7, 103]. Such workflows allow scientists to easily compose existing simulation codes into new applications to be run on large-scale computing platforms. Workflows are often modeled as directed graphs where vertices represent computation tasks of the workflow and edges represent their dependencies. These directed graphs are either acyclic for non-iterative applications or may contain cycles representing iterations. Several management systems have been proposed to tackle the increasing complexity of computational workflows [94]. In Chapter 3, we focus on iterative workflows whose directed graph is a linear chain of parallel tasks. In other words, the same set of tasks is executed repeatedly until the execution completes. This general applicative framework includes all restarted Krylov subspace methods [45] where the same sequence of tasks is executed until convergence. It also addresses Uncertainty Quantification (UQ) workflows, which attempt to close a fundamental gap between simulations and the practical physical systems they represent: simulations based on mathematical modeling are deterministic by nature, while the real-life behavior of the system has natural variability. UQ workflows roughly consist in performing many simulations with slightly different initial conditions in order to properly estimate the uncertainty of the simulation prediction [76, 87]. The large set of simulations to be performed is often organized into phases that are similar to iterations of an iterative application. When designing checkpoint/restart strategies for task-based workflows, it is natural to take checkpoint between the completion of some task and the beginning of its successor. This way, the checkpoint mechanism can be provided by the workflow management system without having to modify the code of each task. However, this restricts the time-steps at which checkpoints can be taken and makes the optimization problem of selecting the best checkpoint times more difficult. Furthermore, the data to checkpoint is now the output of the tasks and may have different sizes for different tasks of the workflow.

Batch schedulers, a.k.a Resource and Job Management Systems (RJMS), are a key component of the supercomputing infrastructure. Users make a reservation for their parallel job that includes information such as an upper-bound on the expected length (called the wall time), and the desired number of resources needed for the execution. Sophisticated scheduling heuristics have been introduced to accommodate for the job submissions on the fly; these heuristics go well beyond the naive *First Come First Served (FCFS)* policy and are designed for the batch scheduler to allocate these jobs on the computing platform, with the end goal of optimizing some metric or combination of metrics. In the last decade, batch schedulers have faced additional constraints: on state-of-the-art platforms, an increasing number of users experience the crash of a node belonging to their reservation set during the execution of their job. This is because platforms are composed of more and more nodes to accommodate an endless increase in job demands. This scaling is the main reason for the increasing number of failures, as stated above. In Chapter 4, we propose a novel approach to schedule the failed job in order to improve the performance of the platform.

Traditional iterative methods for solving sparse linear systems can be divided into two categories: stationary iterative methods, such as the Jacobi method [88], the Gauss-Seidel method [88], and the Successive Overrelaxation (SOR) method [41, 108], or non-stationary iterative methods, such as Krylov subspace methods including Conjugate Gradient (CG) method [56, 66], Generalized Minimal Residual (GMRES) method [89], Generalized Conjugate Residual (GCR) method [37], etc. According to different iterative methods, we abstract them into various iterative applications. In Chapter 2, we focus on stochastic iterative applications for the stationary iterative method. The stochastic iterative applications can be considered as a linear chain whose tasks do not have constant execution times but obey some probability distributions. While in Chapter 3, we focus on deterministic iterative applications, such as the one for the Krylov subspace method. Deterministic iterative applications can be considered as a cyclic chain of tasks, the duration of each task of an iteration is the same among all iterations.

The rest of the thesis is organized as follows: In Chapter 1, we review the related work of this thesis. In Chapter 2, we extend the results of Toueg and Babaoglu [105] to deal with linear chains whose tasks do not have constant execution times but instead obey some probability distributions. After that, in Chapter 3, we provide a general-purpose approach to deal with fail-stop errors in iterative applications. Our optimal checkpointing strategy is agnostic of any specific property of the target iterative application. Instead, it abstracts the iterative application as a chain of cyclic tasks and provides the optimal periodic checkpoint pattern based only upon general information such as task durations and checkpoint costs. Finally, we study a more practical problem in Chapter 4: if no free node of the platform is available at the time of a failure, how to efficiently schedule the failed job, which should wait until enough resources become available for its re-execution. The main contributions of each chapter are summarized below.

Chapter 1: Related work

This is a preliminary chapter in which we introduce related work. We introduce checkpointing, iterative applications, fault-tolerance methods for iterative applications, batch schedulers, and general fault-tolerance mechanisms, respectively.

Chapter 2: Stochastic iterative applications [C1, R2]

As stated above, the Young/Daly formula for periodic checkpointing is known to hold for a divisible load application where one can checkpoint at any time-step. In this chapter, we assess the accuracy of the formula for applications decomposed into computational iterations where: (i) the duration of an iteration is stochastic, i.e., obeys a probability distribution law $\mathcal{D}$ of mean $\mu_{\mathcal{D}}$; and (ii) one can checkpoint only

at the end of an iteration. We first consider static strategies where checkpoints are taken after a given number of iterations k and provide a closed-form, asymptotically optimal formula for k, valid for any distribution $\mathcal{D}$. We then show that using the Young/Daly formula to compute k (as $k \cdot \mu_{\mathcal{D}} = \mathcal{P}_{YD}$) is a first-order approximation of this formula. We also consider dynamic strategies where one decides to checkpoint at the end of an iteration only if the total amount of work since the last checkpoint exceeds a threshold W_{th} , and otherwise proceed to the next iteration. Similarly, we provide a closed-form formula for this threshold and show that $\mathcal{P}_{YD}$ is a first-order approximation of W_{th} . Finally, we provide an extensive set of simulations where $\mathcal{D}$ is either Uniform, Gamma, or truncated Normal, which shows the global accuracy of the Young/Daly formula, even when the distribution $\mathcal{D}$ had a large standard deviation (and when one cannot use a first-order approximation). Hence we establish that the relevance of the formula goes well beyond its original framework.

Chapter 3: Deterministic iterative applications [J3, R1]

After studying the fault-tolerance methods for stochastic iterative applications in Chapter 2, in this following chapter, we provide an optimal checkpointing strategy to protect iterative applications from fail-stop errors. We consider a general framework where the application repeats the same execution pattern by executing consecutive iterations and where each iteration is composed of several tasks. These tasks have different execution lengths and different checkpoint costs. The first naive strategy would checkpoint after each task. The second naive strategy would checkpoint at the end of each iteration. The third strategy inspired by the Young/Daly formula would work for $\sqrt{2\mu c_{ave}}$ seconds, where μ is the application MTBF and c_{ave} is the average checkpoint time, and checkpoint at the end of the current task (and repeat). The fourth strategy is a periodic extension of Young/Daly approach: it chooses the task of an iteration with minimum checkpoint size. Only the result of this task will (possibly) be checkpointed. Then it uses the Young-Daly formula to compute how many iterations to include in between two checkpoints. All these naive and Young/Daly strategies are suboptimal. Our main contribution is to show that the optimal checkpoint strategy is globally periodic and to design a dynamic programming algorithm that computes the optimal checkpointing pattern. This pattern may well checkpoint many different tasks, and this across many different iterations. We show through simulations, both from synthetic and real-life application scenarios, that the optimal strategy outperforms the naive and Young/Daly strategies.

Chapter 4: Node stealing for failed jobs

After a machine failure, batch schedulers typically re-schedule the failed job with a high priority. It is fair for the failed job but still requires that job to re-enter the submission queue and to wait for enough resources to become available. The waiting time can be very long when the job is large and the platform highly loaded, as is the case with typical HPC platforms. We propose another strategy: when a job J fails, if no platform node is available, we steal one node from another job J', and use it to continue the execution of J despite the failure. Thus, job J' is killed and resubmitted later, waiting for enough available resources to re-execute. In this chapter, we give a detailed assessment of this node stealing strategy using traces from the Mira supercomputer at Argonne National Laboratory. The main conclusion is that node stealing improves the utilization of the platform and dramatically reduces the flow of large jobs at the price of slightly increasing the flow of small jobs.

Résumé français

En raison du développement rapide des technologies de l'information, les ordinateurs sont largement utilisés dans tous les domaines de la science et de la technologie modernes. Le calcul scientifique a toujours été un domaine d'application informatique important. Avec l'augmentation de la mémoire des ordinateurs et la vitesse de calcul, la taille des problèmes à résoudre augmente également. Ces dernières années, le développement rapide des ordinateurs à hautes performances, également appelés super-calculateurs, a apporté de nombreux nouveaux défis. En février 2014, dix défis ont été identifiés pour réaliser le système Exascale par l'Advanced Scientific Computing Advisory Committee, dont l'un est la tolérance aux pannes [70].

Le déploiement d'applications scientifiques à grande échelle nécessite des mécanismes de tolérance aux pannes, qui peuvent atténuer l'impact des erreurs et assurer une exécution correcte et ininterrompue de l'application [19, 20]. Mais du point de vue de la tolérance aux pannes, la difficulté vient du passage à l'échelle. Des supercalculateurs comme Frontier, Fugaku ou LUMI (respectivement classés 1er, 2e et 3e du classement TOP500 [104]) embarquent désormais des millions de cœurs (avec un pic à 10,6M pour Sunway TaihuLight (6e)). Ces grands systèmes informatiques sont fréquemment confrontés à des pannes, également appelées erreurs fatales (telles que des pannes matérielles). Même si chacun de leurs cœurs a une très faible probabilité de défaillance, la probabilité de défaillance de l'ensemble du système est beaucoup plus élevée. Plus précisément, supposons que le temps moyen entre chaque défaillance (*Mean Time Between Failure* ou MTBF) de chaque ressource de calcul soit d'environ 10 ans, ce qui signifie qu'une telle ressource ne devrait connaître une erreur que tous les dix ans en moyenne, et qui explique pourquoi les ressources de calcul sont individuellement très fiables. Lors de l'exécution d'un code de simulation sur 100 000 de ces ressources en parallèle, le MTBF est réduit à seulement 50 minutes [55]: en moyenne, un nœud de la plate-forme de calcul subit une panne toutes les 50 minutes. Avec un million de ces ressources, le MTBF est réduit à cinq minutes, tandis que les codes déployés sur ces plates-formes à grande échelle durent généralement des heures ou des jours. À mesure que la demande de puissance de calcul augmente, les pannes ne peuvent plus être ignorées et des mécanismes de tolérance aux pannes doivent être déployés.

La manière classique de gérer les défaillances dans les systèmes informatiques à grande échelle consiste en des mécanismes de point de contrôle/retour en arrière. Un *point de contrôle* de l'application est pris périodiquement, c'est-à-dire que l'état de l'application (généralement tout le contenu de sa mémoire) est écrit sur un stockage fiable. Chaque fois que l'une des ressources informatiques rencontre une panne, l'application s'interrompt et redémarre à partir du dernier point de contrôle valide. Plusieurs études se sont penchées sur la question cruciale de la période de contrôle optimale, définie comme le temps entre deux points de contrôle consécutifs. D'une part, si les points de contrôle sont pris trop souvent, du temps est perdu dans des opérations d'E/S coûteuses. D'un autre côté, si les points de contrôle sont trop peu fréquents, du temps sera perdu à recalculer de grandes parties de l'application après chaque panne. Fait intéressant, la fiabilité était déjà une question au début de l'informatique: dans les années 70, Young a proposé une approximation du premier ordre du temps optimal entre deux points

de contrôle qui minimise l'espérance de la durée de l'ensemble du calcul [110]. L'approximation de Young a ensuite été affinée par Daly trente ans plus tard [29]. Young [110] et Daly [29] ont ainsi obtenue la formule bien connue de Young/Daly $\mathcal{P}_{YD} = \sqrt{2\mu_f C}$ pour la période de point de contrôle optimale, où μ_f est le MTBF de la plate-forme et C est la durée du point de contrôle. En supposant une vitesse unitaire, le temps $\mathcal{P}_{YD}$ écoulé entre deux points de contrôle est également la quantité de travail exécuté pendant chaque période. La formule Young/Daly s'applique aux applications où un point de contrôle peut être pris à tout moment pendant le calcul. Les applications divisibles [13, 85] sont un exemple de telles applications. Aussi, la formule Young/Daly suppose que le temps nécessaire pour prendre un point de contrôle est constant (ce qui correspond à une taille constante des données à sauvegarder).

Cependant, de nombreuses applications scientifiques présentent un comportement plus compliqué. Dans le chapitre 2, nous nous concentrons sur les *applications itératives* que nous définissons comme des applications composées de nombreuses *itérations* de calcul, où l'on ne peut prendre un point de contrôle qu'à la fin d'une itération. En effet, pour les applications itératives, le point de contrôle n'est efficace (ou même possible) qu'à la fin d'une itération, car le volume de données à copier est considérablement réduit à ce stade. Un large éventail d'applications s'inscrit dans ce cadre. Les solveurs itératifs pour les systèmes d'algèbre linéaire creux en sont un exemple représentatif [83, 88]. De plus, le temps de chaque itération dépend de plusieurs paramètres (modèle de parcimonie de certains vecteurs, conflit de communication, performances du système) et peut varier considérablement d'une itération à l'autre. Ce phénomène est amplifié dans les méthodes itératives randomisées [46] où des vecteurs aléatoires sont générés au fur et à mesure que l'application progresse. Une autre classe d'applications qui se décompose naturellement en itérations de longueur variable sont les applications *Bulk Synchronous Parallel* (BSP) [44, 57] où l'on effectue les points de contrôle à la fin de chaque opération de jointure. Un exemple typique d'une séquence BSP d'opérations de *fork-join* est le calcul à n corps [15]. En raison de la simplicité du modèle de programmation, de nombreuses applications BSP sont déployées à grande échelle [14].

Les flux de travail scientifiques (ou *workflows*) représentent une grande partie des applications complexes déployées sur les superordinateurs et couvrent un large éventail de domaines, tels que la prévision météorologique, la modélisation du climat, l'astronomie et la bioinformatique [7, 103]. Ces flux de travail permettent aux scientifiques de composer facilement des codes de simulation existants en de nouvelles applications à exécuter sur des plates-formes informatiques à grande échelle. Ces flux de travail sont souvent modélisés sous forme de graphes orientés où les sommets représentent les tâches de calcul et les arêtes représentent leurs dépendances. Ces graphes orientés sont soit acycliques pour les applications non itératives, soit peuvent contenir des cycles représentant des itérations. Plusieurs systèmes de gestion ont été proposés pour faire face à la complexité croissante des workflows informatiques [94].

Dans le chapitre 3, nous nous concentrons sur les workflows itératifs dont le graphe orienté est une chaîne linéaire de tâches parallèles. En d'autres termes, le même ensemble de tâches est exécuté à plusieurs reprises jusqu'à la fin de l'exécution. Ce cadre applicatif général inclut toutes les méthodes de sous-espace de Krylov redémarrées [45] où la même séquence de tâches est exécutée jusqu'à convergence. Il aborde également les workflows de quantification de l'incertitude (UQ), qui tentent de combler un fossé fondamental entre les simulations et les systèmes physiques pratiques qu'elles représentent: les simulations basées sur la modélisation mathématique sont déterministes par nature, tandis que le comportement réel du système présente une variabilité naturelle. Les workflows UQ consistent en gros à effectuer de nombreuses simulations avec des conditions initiales légèrement différentes afin d'estimer correctement l'incertitude de la prédiction de la simulation [76, 87]. Le grand nombre de simulations à effectuer est souvent organisé en phases qui s'apparentent aux itérations d'une application itérative.

Lors de la conception de stratégies de point de contrôle/redémarrage pour les flux de travail basés sur des tâches, il est naturel de prendre un point de contrôle entre l'achèvement d'une tâche et le début

de son successeur. De cette façon, le mécanisme de point de contrôle peut être fourni par le système de gestion de workflow sans avoir à modifier le code de chaque tâche. Cependant, cela limite les pas de temps auxquels les points de contrôle peuvent être pris et rend plus difficile le problème d'optimisation de la sélection des meilleurs moments des points de contrôle. De plus, les données à contrôler sont désormais la sortie des tâches et peuvent avoir des tailles différentes pour différentes tâches du flux de travail.

Les planificateurs par lots, également appelés systèmes de gestion des ressources et des tâches (RJMS), sont un élément clé de l'infrastructure de calcul intensif. Les utilisateurs effectuent une réservation pour leur travail parallèle qui inclut des informations telles qu'une limite supérieure sur la durée attendue (appelée le *wall time*) et le nombre souhaité de ressources nécessaires à l'exécution. Des heuristiques de planification sophistiquées ont été introduites pour s'adapter aux soumissions de travaux à la volée ; ces heuristiques vont bien au-delà de la politique naïve *First Come First Served (FCFS)* et sont conçues pour que le planificateur de lots alloue ces tâches sur la plate-forme informatique, dans le but final d'optimiser une métrique ou une combinaison de métriques.

Au cours de la dernière décennie, les ordonnanceurs par lots ont été confrontés à des contraintes supplémentaires: sur les plates-formes de pointe, un nombre croissant d'utilisateurs subissent le crash d'un nœud appartenant à leur ensemble de réservation lors de l'exécution de leur travail. En effet, les plates-formes sont composées de plus en plus de nœuds pour répondre à une augmentation sans fin des demandes de travail. Ce passage à l'échelle est la principale raison du nombre croissant d'échecs, comme indiqué ci-dessus. Dans le chapitre 4, nous proposons une nouvelle approche pour planifier la tâche ayant échoué afin d'améliorer les performances de la plateforme.

Le reste de la thèse est organisé comme suit: dans le chapitre 1, nous passons en revue la littérature portant sur les thématiques de cette thèse. Dans le chapitre 2, nous étendons les résultats de [105] pour traiter des chaînes linéaires dont les tâches n'ont pas de temps d'exécution constants mais obéissent à des distributions de probabilité. Après cela, dans le chapitre 3, nous fournissons une approche générale pour traiter les erreurs fatales dans les applications itératives. Notre stratégie de point de contrôle optimale est indépendante de toute propriété spécifique de l'application itérative cible. Au lieu de cela, elle abstrait l'application itérative en une chaîne de tâches cycliques et fournit le modèle de point de contrôle périodique optimal basé uniquement sur des informations génériques telles que la durée des tâches et les coûts des points de contrôle. Enfin, nous étudions un problème plus pratique dans le chapitre 4: si aucun nœud libre de la plate-forme n'est disponible au moment d'une panne, comment planifier efficacement le travail en échec qui doit attendre que suffisamment de ressources soient disponibles pour sa réexécution? Les principales contributions de chaque chapitre sont résumées ci-dessous.

Chapitre 1: Etat de l'art

Il s'agit d'un chapitre préliminaire dans lequel nous introduisons les travaux connexes. Nous introduisons respectivement les points de contrôle, les applications itératives, les méthodes de tolérance aux pannes pour les applications itératives, les ordonnanceurs par lots et les mécanismes généraux de tolérance aux pannes.

Chapitre 2: Applications itératives stochastiques [C1, R2]

Comme indiqué ci-dessus, la formule de Young/Daly pour les points de contrôle périodiques est connue pour être valable pour une application de charge divisible où l'on peut effectuer un point de contrôle à n'importe quel pas de temps. Dans ce chapitre, nous évaluons la précision de la formule pour des applications décomposées en itérations de calcul où: (i) la durée d'une itération est stochastique, c'est-à-dire obéit à une loi de distribution de probabilité $\mathcal{D}$ de moyenne $\mu_{\mathcal{D}}$; et (ii) on ne peut prendre un

point de contrôle qu'à la fin d'une itération. Nous considérons d'abord des stratégies statiques où les points de contrôle sont pris après un nombre donné d'itérations k et fournissons une formule close, asymptotiquement optimale, pour k, valable pour toute distribution $\mathcal{D}$. Nous montrons ensuite que l'utilisation de la formule de Young/Daly pour calculer k (as $k \cdot \mu_{\mathcal{D}} = \mathcal{P}_{YD}$) est une approximation au premier ordre de cette formule. Nous considérons également des stratégies dynamiques où l'on décide de ne prendre un point de contrôle à la fin d'une itération que si la quantité totale de travail depuis le dernier point de contrôle dépasse un seuil W_{th} , et sinon de passer à l'itération suivante. De même, nous fournissons une formule close pour ce seuil et montrons que $\mathcal{P}_{YD}$ est une approximation au premier ordre de W_{th}. Enfin, nous fournissons un ensemble complet de simulations où $\mathcal{D}$ suit soit une distribution uniforme, gamma ou normale tronquée, ce qui montre la précision globale de la formule Young/Daly, même lorsque la distribution $\mathcal{D}$ a un grand écart type (et quand on ne peut pas utiliser une approximation du premier ordre). Nous constatons ainsi que la pertinence de la formule dépasse largement son cadre d'origine.

Chapitre 3: Applications itératives déterministes [J3, R1]

Après avoir étudié les méthodes de tolérance aux pannes pour les applications itératives stochastiques dans le chapitre 2, dans ce chapitre suivant, nous fournissons une stratégie de point de contrôle optimale pour protéger les applications itératives des erreurs fatales. Nous considérons un cadre général, où l'application répète le même modèle d'exécution en exécutant des itérations consécutives, et où chaque itération est composée de plusieurs tâches. Ces tâches ont des durées d'exécution différentes et des coûts de point de contrôle différents. Supposons qu'il y a n tâches et que cette tâche a_i, où $0 \leq i < n$, a un temps d'exécution t_i et un coût de point de contrôle c_i. Une stratégie naïve serait un point de contrôle après chaque tâche. Une autre stratégie naïve serait un point de contrôle à la fin de chaque itération. Une stratégie inspirée de la formule Young/Daly fonctionnerait pendant $\sqrt{2\mu c_{ave}}$ secondes, où μ est le MTBF de l'application et c_{ave} est le temps de point de contrôle moyen, et prendrait un point de contrôle à la fin de la tâche en cours (et ainsi de suite). Une autre stratégie, également inspirée de la formule Young/Daly, sélectionnerait la tâche $a_{\min}$ avec le plus petit coût de point de contrôle $c_{\min}$ et ferait un point de contrôle après chaque p^{th} instance de cette tâche, conduisant à une période de point de contrôle pT, où $T = \sum_{i=0}^{n-1} a_i$ est le temps par itération. On choisirait la période pour que $pT \approx \sqrt{2\mu c_{\min}}$ obéisse à la formule de Young/Daly. Toutes ces stratégies naïves et/ou inspirées par Young/Daly sont sous-optimales. Notre principale contribution montre que la stratégie de point de contrôle optimale est globalement périodique propose un algorithme de programmation dynamique qui calcule le modèle de point de contrôle optimal. Ce modèle prend des points de contrôle après des tâches potentiellement différentes après des nombres d'itérations potentiellement différents. Nous montrons par des simulations à partir de scénarios d'application synthétiques et réels que la stratégie optimale surpasse les stratégies naïve et à la Young/Daly.

Chapitre 4: Vol de nœud pour les travaux ayant échoué

Après une panne de machine, les planificateurs de lots replanifient généralement le travail qui a échoué avec une priorité élevée. Ceci est acceptable pour le travail ayant échoué, mais nécessite toujours que ce travail réintègre la file d'attente de soumission et attende que suffisamment de ressources soient disponibles. Le temps d'attente peut être très long lorsque le travail est volumineux et que la plate-forme très chargée, comme c'est souvent le cas avec les plates-formes de calcul haute-performance. Nous proposons une autre stratégie : lorsqu'un travail J échoue, si aucun nœud de plate-forme n'est disponible, nous volons un nœud de calcul attribué à un autre travail J', et l'utilisons pour continuer l'exécution de J malgré son échec. Dans ce chapitre, nous donnons une évaluation détaillée de cette stratégie de

vol de nœuds en utilisant les traces du super-calculateur Mira du Laboratoire National d'Argonne aux Etats-Unis. La principale conclusion est que le vol de nœud améliore l'utilisation de la plate-forme et réduit considérablement le temps de calcul des gros travaux, au prix d'une légère augmentation du temps de calcul des petits travaux.

Chapter 1

Related work

We survey the previous work and the related work of this thesis in this section. At first, we overview checkpointing in Section 1.1. Then we discuss iterative applications in Section 1.2, including stochastic iterative applications in Section 1.2.1 and deterministic iterative applications in Section 1.2.2. Furthermore, we present fault-tolerance methods for iterative applications in Section 1.3. As for the work in Chapter 4, we give a few pointers to related work on batch schedulers in Section 1.4. Finally, we end with fault-tolerance mechanisms in Section 1.5.

1.1 Checkpointing

Checkpoint-restart is one of the most used strategies to deal with fail-stop errors, and several variants of this policy have been studied, see [55] 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 divisible-load application, results were first obtained by Young [110] and Daly [29], who showed how to derive the optimal checkpointing period. Given the checkpointing cost C and platform MTBF μ, the classical formula due to Young [110] and Daly [29] states that the optimal checkpointing period is $P_{YD} = \sqrt{2\mu C}$. This periodic strategy has been extended to deal with a multi-level checkpointing scheme [12, 31, 72], or by using SSD or NVRAM as secondary storage [20].

Going beyond divisible-load applications, some works target checkpointing strategies for workflows. Workflows are expressed in terms of directed acyclic graphs (DAGs) where vertices represent the computational tasks and edges represent dependences between tasks. Workflows are similar to iterative applications where checkpointing is only possible right after the completion of a task. In fact, the simplest workflows are linear workflows, i.e., applications that can be expressed as a linear chain of (parallel) tasks. If these tasks are parallel, we have an iterative application whose iterations have deterministic execution times, namely the durations of the tasks. Checkpointing is only possible right after the completion of a task, and the problem is to determine which tasks should be checkpointed. The problem of finding the optimal checkpoint strategy for a linear chain of tasks (determining which tasks to checkpoint), in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [105] using a dynamic programming algorithm. We stress that this latter approach is not suited to iterative applications. The main limitation of using results for computational workflows is the complexity. Solutions such as that of Toueg and Babaoglu [105] use the whole description of the graph. Chapter 3 shows that we do not need that much information by proving that we can focus on periodic solutions (as is the case for divisible load applications) and that the size of the period can be bounded. Indeed, consider an iterative application with a large number of iterations, say $N_{iter} = 10,000$ iterations, and assume that

$n = 10$ (10 tasks per iteration). One solution to finding an optimal checkpointing strategy could be: (i) unroll the loop and build a linear chain of $n \times N_{iter} = 100,000$ tasks; (ii) apply the algorithm of [105] to this huge chain and return the optimal solution. However, the cost of this algorithm is quadratic in the value of N_{iter} . Worse, if we re-execute the same application for $N_{iter} = 20,000$ iterations, we have to recompute the optimal solution from scratch. On the contrary, our approach in Chapter 3 provides a generic and compact solution that does not depend upon the value of N_{iter} .

For general workflows, finding an optimal solution is a #P-complete problem [50]. Recall that #P is the class of counting problems that correspond to NP decision problems [106], and that #P-complete problems are at least as hard as NP-complete problems. Several heuristics to decide which tasks to checkpoint are proposed and evaluated in [51]. As pointed out above, for general workflows, deciding which tasks to checkpoint has been shown #P-complete [50], but the results of [8] show that if the graph is scheduled in a sequential manner (linearized), then one can derive an optimal checkpointing strategy. In Chapter 3, we focus on pipelined linear workflows, i.e., on applications expressed as a linear chain of tasks that repeats iteratively.

1.2 Iterative applications

1.2.1 Stochastic iterative applications

Iterative methods are popular for solving large sparse linear systems, which have a wide range of applications in several scientific and industrial problems. Iterative methods fall into two main categories: stationary iterative methods and non-stationary iterative methods. Stationary iterative methods include many classic iterative methods, like the Jacobi method [88], the Gauss-Seidel method [88] and the Successive Overrelaxation (SOR) method [41, 108]. In recent years, randomized iterative methods have been much more popular. For example, the randomized Kaczmraz method [99] and the greedy randomized Kaczmarz method [10] for solving the consistent linear system, the randomized coordinate descent method [68, 71] and the greedy randomized coordinate descent method [9] for solving the least squares problems. For these iterative methods, it is economical to set checkpoints at the end of the iterations since the volume of data that needs to be stored is dramatically reduced at that point. Furthermore, in all these methods, the time spent per iteration is not constant: for classic iterative methods, the amount of flops is usually the same per iteration, but the communication volume and the amount of contention vary from one iteration to another. The variation becomes more important for randomized applications, where random vectors are generated as the application progresses and the amount of flops per iteration changes according to the sparsity pattern [46].

Another class of iterative applications arises from the Bulk Synchronous Parallel (BSP) model, which was originally suggested as a possible 'bridging' model to serve as a standard interface between the architecture levels and language in parallel computations [44, 57]. The representative n-body computations [15] have a number of important applications in fields such as molecular dynamics, fluid dynamics, computer graphics, and even astrophysics [52]. A BSP computation consists of a sequence of parallel super-steps composed of fork-join operations with independent threads executed in parallel. It is economical to set up checkpoints at the end of the super-steps, which naturally fit the definition of iterations. BSP applications that are deployed at scale [14] are composed of a large number of super-steps whose lengths are data-dependent and can adequately be modeled as drawn from some probability distribution.

1.2.2 Deterministic iterative applications

As mentioned above, the other class for solving large sparse linear systems is non-stationary iterative methods, also known as Krylov subspace methods, including Full Orthogonalization Method (FOM) [88], Conjugate Gradient (CG) method [56, 66], Conjugate Residual (CR) method [40], Generalized Minimal Residual (GMRES) method [89], Minimum residual (MINRES) method [81], Biconjugate Gradient (BiCG) method [66], Bi-conjugate Gradient Stabilized (BiCGSTAB) method [49], Conjugate Gradient Squared (CGS) method [98], Quasi-minimal Residual (QMR) method [42], Generalized Conjugate Residual (GCR) method [37], together with their algorithm-based fault-tolerance (ABFT) variants [4, 67, 111].

The common way to accelerate the convergence is to adopt a restart strategy [88, 95, 109], that is, to fix a small value n (usually much less than the dimension of the sparse matrix). If the n-th iteration does not lead to convergence, then the approximate solution of the last iteration is used as the initial approximate solution of a new iteration, and the Krylov subspace method is restarted. The process is repeated until a satisfactory approximate solution is found. Krylov subspace methods fit our model perfectly in Chapter 3: the outer loop after each restart corresponds to a sequence of iterations, and the inner loop as increasing Krylov subspace within each outer loop corresponds to a sequence of tasks. Since the restarted Krylov subspace method fixes the subspace dimension, each iteration contains the same sequence of tasks. In the Krylov subspace method, it is assumed that checkpoints can only be inserted after each task. The goal of an optimal checkpoint strategy is to find out which tasks should be checkpointed to minimize the expected execution time. See Algorithm 4 in Section 3.4.1 for an illustration of GCR(n) method, with an inner loop of n tasks. We report experiments for the GCR(n) method in Section 3.4.4.

The class of iterative applications goes well beyond sparse linear solvers. Uncertainty Quantification (UQ) workflows explore a parameter space in an iterative fashion [76, 87]. This class also encompasses many image and video processing software that operate a chain of computational kernels (each being a task) on a sequence of data sets (each corresponding to an iteration). Examples include image analysis [93], video processing [47], motion detection [64], signal processing [27, 54], databases [24], molecular biology [86], medical imaging [48], and various scientific data analyses, including particle physics [30], earthquake [63], weather and environmental data analyses [86].

1.3 Fault-tolerance methods for iterative applications

The literature devoted to the study of fault-tolerance methods for iterative linear solvers can be divided into two categories, depending upon whether the focus is on soft errors or on fail-stop errors.

There are some works dealing with soft errors. Soft errors are caused by minimum voltage, radiation or thermal cycling, etc., which may not be immediately detected. The execution is not interrupted, but the output is erroneous. Chen presented online-ABFT in [26], a technique that can detect soft errors in the specific Krylov subspace iterative methods by leveraging the orthogonality relationship of two vectors in the middle of the program execution. For general iterative methods, Tao et al. [102] presented a new online-ABFT approach to detect and recover soft errors by combining a novel checksum-based encoding scheme with a checkpoint/rollback scheme. According to the specific properties of GMRES algorithm, Bridges et al. [17] and Elliott et al. [38] proposed the FT-GMRES algorithm using selective reliability. Similarly, Sao and Vuduc [90] proposed the self-stabilizing CG method in view of the special properties of CG algorithm: they check that orthogonality is preserved by recomputing scalar products that should be zero and restarting whenever a threshold is exceeded. Agullo et al. [3] studied the sensitivity and robust numerical detection for CG method. Ozturk et al. [77] proposed a decreasing energy norm based

on the mathematical properties to detect soft errors leading to silent data corruption (SDC) for GMRES, CG and CR methods. Jaulmes et al. [61, 62] exploited asynchrony from exact forward recovery for Detected and Uncorrected Errors (DUE) in iterative solvers. In addition, Casas et al. [22] presented an approach to improve the resilience of scientific applications with soft errors and applied it to Algebraic Multi-Grid (AMG) algorithm.

There are some works dealing with fail-stop errors. Fail-stop errors are caused by a power loss or oxide wearout, etc., which interrupt the execution immediately, and all contents in the memory are lost. To reduce the fault tolerance overhead incurred by checkpointing, Chen [25] proposed a recovery method for iterative methods without checkpointing based on the specific properties of iterative methods. Tao et al. [101] improved the checkpointing performance for iterative methods under a novel lossy checkpointing scheme. Langou et al. [67] presented a lossy approach which is a checkpoint-free fault tolerant scheme for parallel iterative methods. The iterative method is restarted with a new vector which is a new approximate solution recovered from a fail-stop error by using the data of the non-failed processors. Agullo et al. [4, 111] extended this approach by computing a well-suited initial guess which is defined by interpolating the lost entries of the current iterate vector available on surviving nodes, in order to restart the Krylov method. Pachajoa et al. [78] compared the exact state reconstruction (ESR) approach based on the method proposed by Chen [25] with the heuristic linear interpolation (LI) approach by Langou et al. [67] and Agullo et al. [4, 111]. They later extended the ESR approach for protecting the preconditioned CG method against multiple and simultaneous node failures [79, 80]. Agullo et al. studied the resilience in numerical algorithms [2], a parallel sparse hybrid (direct/iterative) linear solver [6] and eigensolvers [5]. Benacchio et al. [11] investigated the fault tolerance iterative solvers and their application in numerical weather and climate prediction. Altogether, fault-tolerance methods proposed to mitigate the impact of fail-stop errors in iterative applications are application-specific and can only be applied to a particular class of iterative algorithms. Moreover, their performance highly depends upon the specific properties of the algorithms. For instance, it considerably varies from one Krylov method to another.

Although there are some fault-tolerance methods proposed for iterative applications, most of them are based on the algorithmic level, and the performance depends highly on the specific properties of the algorithms. Our work in Chapter 2 firstly shows Young/Daly formula can be safely applied to stochastic iteration applications and Chapter 3 is the first approach (to the best of our knowledge) that (i) is not based on the specific properties of the iterative algorithms; (ii) can be applied to any Krylov subspace methods; (iii) provides a polynomial-time algorithm to compute the optimal checkpoint strategy. As a result, we propose a fault-tolerance method for iterative applications not based on the properties of the Krylov subspace methods but abstracts it into an iterative application with cyclic tasks.

1.4 Job management on HPC platforms

Resource and Job Management Systems (RJMS), a.k.a. Batch schedulers, are intermediary software layers generally managed by a system administrator (examples include Slurm [97], Moab/Maui [60], OAR [18] etc.). This software is in charge of allocating the different jobs through a scheduling heuristic while taking into account various constraints. The most important constraints that a batch scheduler has to account for are the *estimation* by users of the resources needed for a job, both in a spatial dimension (number of processing units) and in a temporal dimension (estimated processing time).

Natural developments in batch scheduling have included more dimensions to the scheduling heuristic, such as heterogeneity of computing resources, fairness to deal with the disparity of job requirements and usage, etc. The scheduling heuristics are typically implemented by the introduction of specific

queues, where jobs with similar characteristics (size, reservation length, priority, ...) are grouped together into the same queue [100]. Each queue is configured with a specific scheduling heuristic.

There are several main scheduling heuristics used for batch scheduling. The default for most schedulers is the First-Come-First-Served (FCFS) policy [60, 96]. This strategy is often tweaked by including the time of arrival (i.e. "first-come" condition) into a more general priority-based, greedy heuristic that includes a wide range of parameters [96]. Other common strategies exist, such as Smallest Job First [53], known to be efficient with respect to the response time objective. The advantage of these greedy heuristics is their low scheduling cost. Their drawback is that it is a less efficient solution with a lot of idle time for the platform. To mitigate this limitation, these heuristics are coupled with a *backfilling* strategy. Backfilling consists of scheduling small jobs in the gaps created by the scheduling solutions. The two main flavors of backfilling are *conservative* (no job in the queue can be delayed by a backfilled job) and EASY (the first job in the queue is never delayed by backfilled jobs) [73]. Chapter 4 focuses on using the conservative approach, but we expect that using EASY or other approaches would lead to very similar results and conclusions.

1.5 Fault-tolerance from a system perspective

To mitigate the impact of node crashes, several techniques are considered, such as replication and checkpointing. In Chapter 4, we consider the de-facto standard approach for HPC, periodic checkpointing [55]. With this technique, users are invited to checkpoint their jobs periodically, with the idea that if a node crashes during execution, then the job will be able to resume from the last checkpoint instead of resuming from scratch. A key advantage of checkpointing is to decrease the amount of re-executed work after a crash. 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, i.e., an application that can be checkpointed at any time-step, the classical formula for computing the optimal checkpointing period is given by Young [110] and Daly [29] as mentioned in Section 1.1.

However, there are several complications related to deciding when and on which resources the job will be allowed to resume execution after experiencing the loss of one node. Several batch schedulers [91] will reschedule a failed job with high priority, thereby enabling an immediate re-execution if there is a free node available. The high priority allows the failed job to avoid a long wait in the job submission queue. Without priority, the delay between the interruption of a job and the beginning of its re-execution is called the resubmission time. Its value typically ranges from several hours to several days if the platform is over-subscribed (up to 10 days for large jobs on the K-computer [107]).

Hori et al. [58] discussed how one can use spare nodes to restart an application that has experienced a node failure. This technique could be applied to our case. In [58], spare nodes are reserved and used only in the case of a failure, which enables the failed job to restart as fast as possible. Prabhakaran et al. [84] discussed the limitations of the reservation of spare nodes, which creates a non-negligible overhead. Instead, they study the case where jobs are moldable and/or malleable; when no idle node is available for a failed job to restart, they propose several strategies such as executing the failed job on fewer nodes or taking a node from a malleable job. In contrast, Chapter 4 applies to rigid nodes (neither moldable nor malleable) and never changes the size of the jobs.

The optimization of fault-tolerance techniques often considers a short downtime (also called rejuvenation time) for the failed resources compared to the platform MTBF. This makes sense when one simply needs to reboot the machine that failed. But in the case of a defective component to be replaced, the downtime can last up to one day, because maintenance is operated at a fixed time every day, e.g.

every morning for the K-computer [107]. Our experiments in Chapter 4 aimed at covering the whole range of possible values for the downtime.

Chapter 2

Stochastic iterative applications

2.1 Introduction

In this chapter, we deal with the stochastic iterative applications. As already mentioned in Section 1.2.1, many iterative or BSP applications exhibit iterations of variable length, typically because each iteration is data-dependent. When considering an iterative application, we assume that the length of each iteration is not known a priori, but instead is drawn randomly from some probability distribution $\mathcal{D}$. Again, with unit speed, the length of the iteration is the amount of work within the iteration. The distribution $\mathcal{D}$ is usually acquired by sampling a few executions. In this chapter, we use several usual distributions, such as Uniform, Gamma or Normal.

The main objective of this chapter is to explore whether the Young/Daly formula applies beyond divisible-load applications. To what extent can we use the formula for iterative applications whose length obey a probability distribution $\mathcal{D}$? We first consider static strategies where checkpoints are taken after a given number of iterations k, and we show that using the Young/Daly formula to compute k (as $k \cdot \mu_{\mathcal{D}} = \mathcal{P}_{YD}$) is asymptotically optimal among such strategies, and remains accurate even when the distribution $\mathcal{D}$ had a large standard deviation. Then we consider dynamic strategies where one decides to checkpoint at the end of an iteration only if the total amount of work since the last checkpoint exceeds a threshold W_{th} , and otherwise proceed to the next iteration; we show that an approximation of the optimal value of W_{th} is $\mathcal{P}_{YD}$. Finally, we provide an extensive set of simulations where $\mathcal{D}$ is either Uniform, Gamma or Normal, which shows the global accuracy of the Young/Daly formula and establish that its relevance goes well beyond its original framework.

The main contributions of this chapter are the following:

- For static solutions, we derive a closed-form formula to compute the optimal checkpointing period, and we show that its first-order approximation corresponds to the Young/Daly formula. The derivation is quite technical, and constitutes a major extension of the deterministic case.
- For dynamic solutions, we derive a closed-form formula to compute the threshold at which one decides either to checkpoint or to execute more work, and we show that its first-order approximation also corresponds to the Young/Daly formula. Again, the derivation is complicated and required to use a simplified objective, using the ratio of expectations of actual time over useful time, instead of the expectation of these ratios (see Section 2.4 for details).
- We conduct an extensive set of experiments with classic probability distributions (Uniform, Gamma, Normal) and we conclude that the Young/Daly formula remains accurate and useful in a stochastic setting.

The rest of the chapter is organized as follows. We formally state the model for iterative applications in Section 2.2. Section 2.3 is the core of the chapter to state the static strategy. We state the dynamic

strategy in Section 2.4. Section 2.5 is devoted to simulations. Finally, we conclude and give hints for future work in Section 2.6.

2.2 Framework

We first introduce all model parameters in Section 2.2.1. Then we formally state the optimization problem, as well as the static and dynamic scheduling strategies in Section 2.2.2.

2.2.1 Model

Platform We consider a parallel platform subject to failures. We assume that the failure inter-arrival times follow an Exponential distribution $\textsc{Exp}(\lambda)$ of parameter λ, whose PDF (Probability Density Function) is $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. The MTBF is $\mu_f = \frac{1}{\lambda}$. When hit by a failure, the platform is unavailable during a downtime D.

Application We consider an iterative application composed of n consecutive iterations. The execution time of each iteration is not known before execution but follows a probability distribution $\mathcal{D}$. The execution times of the iterations are thus modeled with random variables $X_1, \ldots, X_n$, where the X_i are IID (independent and identically distributed) variables following $\mathcal{D}$. Finally, we assume that the iterations are deterministic: the second execution for a given iteration has the same duration as the first one, that is to say, two executions of the same iteration take the same time. After each iteration, one can checkpoint the state of the application at a cost of C units of time. In case of a failure, it takes R units of time (after the downtime D) to recover from the last checkpoint.

Expected execution time of a given iteration Consider an iteration of length W; we normalize platform speed so that the application has unit speed; then W also represents the amount of work performed within the iteration. We recall the following result [55, Proposition 1.1]: the expected execution time to perform a work of size W followed by a checkpoint of size C in the presence of failures (Exponential distribution of parameter λ), with a restart cost R and a downtime D is:

$$T_\lambda(W, C, D, R) = \left(\frac{1}{\lambda} + D\right) e^{\lambda R} \left(e^{\lambda(W+C)} - 1\right). \tag{2.1}$$

In Equation 2.1, one assumes that failures can strike during checkpoint and recovery, but not during downtime.

2.2.2 Objective function

Given an iterative application with n iterations, a *solution* is defined as a checkpointing strategy of the form $\mathcal{S} = (\delta_1, \ldots, \delta_n = 1)$ where $\delta_i = 1$ if and only if we perform a checkpoint after the i-th iteration of length X_i. We always checkpoint at the end of the last iteration to save the output data of the application. A solution with $m \leq n$ checkpoints writes $\mathcal{S} = (\delta_1, \ldots, \delta_n)$, with $1 \leq i_1 < i_2 < \cdots < i_m = n$ and $\delta_j = 1 \iff j \in \{i_1, \ldots, i_m\}$. We let $i_0 = 0$ and let $W_j = \sum_{l=i_{j-1}+1}^{i_j} X_l$ denote the work between the j-th checkpoint and the previous one (or the beginning of the execution if $j = 1$).

We are interested in minimizing the total execution time (makespan) of the application. This makespan is given by random variable:

$$MS(\mathcal{S}) = \sum_{j=1}^{m} T_\lambda(W_j, C, D, R).$$

For given values of iteration lengths (the X_i variables), the value of the makespan $MS(\mathcal{S})$ is the expected execution time over all failure scenarios, weighted with their probabilities to happen.

In this work, we present and analyze two different strategies to build a solution. In the *static* strategy, we decide before the execution which iterations to checkpoint. In other words, a static solution does not depend upon the value of the X_i variables, it is determined without knowing the iteration lengths. In that case, the optimization objective is easy to express: it is the expectation $\mathbb{E}[MS(\mathcal{S})]$ of the variable $MS(\mathcal{S})$ over the range of the X_i variables which are IID and follow $\mathcal{D}$. Formally:

$$\mathbb{E}[MS(\mathcal{S})] = \mathbb{E}\left[\sum_{j=1}^{m} T_\lambda(W_j, C, D, R)\right]. \tag{2.2}$$

In Section 2.3, we show how to design a solution that is asymptotically optimal (where the number of iterations n tends to infinity) among all static solutions.

Contrarily to static strategies, *dynamic* strategies decide which iterations to checkpoint on the fly during execution: at the end of each iteration, we add a checkpoint only if the total work since the last checkpoint (or the beginning of the execution if there was no previous checkpoint) exceeds a given threshold. Hence a dynamic solution may well insert different checkpoints for different values of the iteration lengths. Providing a closed-form formula of the expected makespan of a dynamic solution is complicated, because the values of the δ_i are now conditional to the values of the X_i. We circumvent this difficulty by minimizing the slowdown of a solution, where the slowdown is defined as the ratio of the actual execution time over the base time without any checkpoint nor failure. We refer to Section 2.4 for further details.

2.3 Static strategies

This section focuses on static strategies, where checkpoint decisions are made before the execution, based upon application and platform parameters, and do not depend on the actual lengths of the iterations. As stated in Equation 2.2, the objective is to minimize the expected makespan $\mathbb{E}[MS(\mathcal{S})]$.

Given an application with n iterations, static solutions decide which iterations to checkpoint. One can choose a solution to be periodic with period k, i.e., checkpoints are taken every k iterations, namely at the end of iterations number k, $2k$,... until the last iteration (which is always checkpointed by hypothesis, even if its number n is not a multiple of k). A priori, an optimal solution may well not be periodic. However, we prove in Section 2.3.1 that the periodic solution with period k_{static} given below is asymptotically optimal when n is large, and we show in Section 2.3.3 that the first-order approximation of the period length corresponds to the Young/Daly formula.

2.3.1 Asymptotic optimality

We first characterize the expected makespan of a static solution (possibly non-periodic):

Proposition 2.1. *Given a solution $\mathcal{S} = (\delta_1, \dots, \delta_n)$ and its associated m checkpoint indices $i_1 < i_2 < \cdots < i_m = n$, let $k_j = i_j - i_{j-1}$ denote the number of iterations between the $j-1$-th checkpoint (or the beginning of the execution if $j = 1$) and the j-th checkpoint. Define*

$$C_{ind}(k) = \frac{e^{\lambda C}\mathbb{E}[e^{\lambda X}]^k - 1}{k}, \tag{2.3}$$

then the expected makespan is

$$\mathbb{E}[MS(\mathcal{S})] = e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\sum_{j=1}^{m} k_j \cdot C_{ind}(k_j). \tag{2.4}$$

Proof. Recall that $W_j = \sum_{l=i_{j-1}+1}^{i_j} X_l$ in Equation 2.1. We have

$$\begin{aligned}
&\mathbb{E}\left[T_\lambda(W_j, C, D, R)\right] \\
&= \int_{\mathcal{D}_{W_j}} e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\left(e^{\lambda(w+C)}-1\right) f_{W_j}(w)dw \\
&= e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\left(\int_{\mathcal{D}_{W_j}} e^{\lambda(w+C)} f_{W_j}(w)dw - 1\right) \\
&= e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\left(e^{\lambda C}\mathbb{E}[e^{\lambda W_j}]-1\right) \qquad (2.5)\\
&= e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\left(e^{\lambda C}\prod_{i=i_{j-1}+1}^{i_j}\mathbb{E}[e^{\lambda X_i}]-1\right) \qquad (2.6)\\
&= e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\left(e^{\lambda C}\mathbb{E}[e^{\lambda X}]^{i_j-i_{j-1}}-1\right). \qquad (2.7)
\end{aligned}$$

Equation 2.6 holds because the random variables X_i are independent, and Equation 2.7 holds because they are identically distributed. Using the number of iterations $k_j = i_j - i_{j-1}$ included in W_j, we rewrite the expected cost of $\mathcal{S}$ as:

$$\mathbb{E}[MS(\mathcal{S})] = e^{\lambda R}\left(\frac{1}{\lambda}+D\right)\sum_{j=1}^{m}\left(e^{\lambda C}\mathbb{E}[e^{\lambda X}]^{k_j}-1\right).$$

□

Note that $\mathbb{E}[e^{\lambda X}]$ is easy to compute for well-known distributions, and we give examples below. Equation 2.4 provides a closed-form formula to compute the expected makespan of a static solution. Recall that the principal Lambert function $\mathcal{W}_0$ is defined for $x \geq -\frac{1}{e}$ by $\mathcal{W}_0(x) = y$ if $ye^y = x$. The asymptotically optimal solution is given by the following theorem;

Theorem 2.1. *The periodic solution checkpointing every k_{static} iterations is asymptotically optimal, where*

$$x_{static} = \frac{\mathcal{W}_0(-e^{-\lambda C-1})+1}{\log\left(\mathbb{E}[e^{\lambda X}]\right)} \tag{2.8}$$

and k_{static} is either $\max(1, \lfloor x_{static} \rfloor)$ or $\lceil x_{static} \rceil$, whichever achieves the smaller value of $C_{ind}(k)$ (computed by Equation 2.3).

Proof. We first show that the function $C_{\text{ind}}(x)$ reaches its minimum for $x = x_{\text{static}}$:

Lemma 2.1. *The function $x \mapsto C_{ind}(x)$ is decreasing on $[0, x_{static}]$ and increasing on $[x_{static}, \infty)$ where x_{static} is defined by Equation 2.8.*

Proof. We differentiate and study the variations of C'_{ind}. We get

$$C'_{\text{ind}}(x) = \frac{xe^{\lambda C}\log\left(\mathbb{E}[e^{\lambda X}]\right)\mathbb{E}[e^{\lambda X}]^x - \left(e^{\lambda C}\mathbb{E}[e^{\lambda X}]^x - 1\right)}{x^2}.$$

Letting $y = x \log\left(\mathbb{E}[e^{\lambda X}]\right) - 1$, we have $e^y = \mathbb{E}[e^{\lambda X}]^x e^{-1}$ and we obtain

$$\begin{aligned} C'_{\text{ind}}(x) &= \frac{e^{\lambda C} y \mathbb{E}[e^{\lambda X}]^x + 1}{x^2} \\ &= \frac{e^{\lambda C+1} y e^y + 1}{x^2}. \end{aligned}$$

We derive that $C'_{\text{ind}}(x) \leq 0 \Leftrightarrow ye^y \leq -e^{-\lambda C-1}$. The function ye^y is an increasing function of y (and hence of x), and the equality is reached for $y = \mathcal{W}_0(-e^{-\lambda C-1})$ where $\mathcal{W}_0$ is the principal Lambert function. Finally, when $y = \mathcal{W}_0(-e^{-\lambda C-1})$, we have $x = x_{\text{static}}$.

Therefore, the function $C'_{\text{ind}}(x)$ has a unique zero x_{static}, is negative on $[0, x_{\text{static}}]$ and is positive on $[x_{\text{static}}, \infty)$. This shows that the function $C_{\text{ind}}(k)$ for integer values of k reaches its minimum either for $\max(1, \lfloor x_{\text{static}} \rfloor)$ or $\lceil x_{\text{static}} \rceil$, and we retrieve the definition of k_{static}. This concludes the proof of Proposition 2.1. □

We consider Equation 2.3 again and re-write it as

$$\begin{aligned} \mathbb{E}[MS(\mathcal{S})] &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right) \sum_{j=1}^{m} k_j \cdot C_{\text{ind}}(k) \\ &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right) \sum_{k=1}^{\infty} n_k \cdot k \cdot C_{\text{ind}}(k), \end{aligned} \tag{2.9}$$

where n_k is the number of inter-checkpoint intervals with k iterations. We let $n_k = 0$ if there is no interval with k iterations, hence the infinite sum is well-defined.

We now introduce the periodic solution $\mathcal{S}_p$ that checkpoints every k_{static} iterations until the end of the execution, as long as there are at least k_{static} iterations left, and then checkpoints every remaining iterations. Formally, with an Euclidean division, letting $n_{\text{div}} = \lfloor n/k_{\text{static}} \rfloor$ and $n_{\text{mod}} = n \bmod k_{\text{static}}$, we have $n = n_{\text{div}} k_{\text{static}} + n_{\text{mod}}$ and $0 \leq n_{\text{mod}} < k_{\text{static}}$. Hence the solution $\mathcal{S}_p$ has n_{div} intervals of k_{static} iterations, and the few remaining n_{mod} iterations, if any, are checkpointed individually. The expected makespan of $\mathcal{S}_p$ is

$$\begin{aligned} \mathbb{E}[MS(\mathcal{S}_p)] &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\left(n_{\text{div}} k_{\text{static}} C_{\text{ind}}(k_{\text{static}}) + n_{\text{mod}} C_{\text{ind}}(n_{\text{mod}})\right) \\ &\leq e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\left(n C_{\text{ind}}(k_{\text{static}}) + (k_{\text{static}} - 1) C_{\text{ind}}(n_{\text{mod}})\right). \end{aligned}$$

From Equation 2.9, and because $C_{\text{ind}}(k_{\text{static}})$ is minimum over all possible values of k, we get

$$\mathbb{E}[MS(\mathcal{S}_{\text{opt}})] \geq e^{\lambda R}\left(\frac{1}{\lambda} + D\right) n C_{\text{ind}}(k_{\text{static}}).$$

Hence we can bound the ratio as follows:

$$\begin{aligned} \frac{\mathbb{E}[MS(\mathcal{S}_p)]}{\mathbb{E}[MS(\mathcal{S}_{\text{opt}})]} &\leq \frac{n C_{\text{ind}}(k_{\text{static}}) + (k_{\text{static}} - 1) C_{\text{ind}}(n_{\text{mod}})}{n C_{\text{ind}}(k_{\text{static}})} \\ &= 1 + \frac{k_{\text{static}} - 1}{n} \frac{C_{\text{ind}}(n_{\text{mod}})}{C_{\text{ind}}(k_{\text{static}})} \\ &= 1 + O\left(\frac{1}{n}\right). \end{aligned}$$

This shows the asymptotic optimality of solution $\mathcal{S}_p$ and concludes the proof of Theorem 2.1. □

2.3.2 Instantiation for some distribution laws

We recall the definition of some well-known distributions laws that we use for $\mathcal{D}$, and show how to compute x_{static} for each of them.

Uniform law Let X (the random variable for an iteration length) obey an Uniform distribution law $\text{UNIFORM}(a, b)$ on $[a, b]$, where $0 < a < b$. The PDF (Probability Density Function) is $f(x) = \frac{1}{b-a}$ for $x \in [a, b]$. We have $\mu_{\mathcal{D}} = \frac{a+b}{2}$ and $\mathbb{E}[e^{\lambda X}] = \frac{e^{\lambda b} - e^{\lambda a}}{\lambda(b-a)}$, hence $x_{\text{static}} = \frac{\mathcal{W}_0(-e^{-\lambda C-1})+1}{\log\left(\frac{e^{\lambda b}-e^{\lambda a}}{\lambda(b-a)}\right)}$.

Gamma law Let X obey a Gamma law $\text{GAMMA}(\alpha, \beta)$, where $\alpha, \beta > 0$. The PDF is $f(x) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$ for $x \geq 0$. We have $\mu_{\mathcal{D}} = \frac{\alpha}{\beta}$ and $\mathbb{E}[e^{\lambda X}] = \left(\frac{\beta}{\beta-\lambda}\right)^\alpha$, hence $x_{\text{static}} = \frac{\mathcal{W}_0(-e^{-\lambda C-1})+1}{\alpha \log\left(\frac{\beta}{\beta-\lambda}\right)}$. Note that a Gamma law $\text{GAMMA}(1, \beta)$ is an Exponential law of parameter β.

Normal law Let X obey a Normal law $\text{NORMAL}(\mu, \sigma^2)$, where $\mu, \sigma > 0$. The PDF is $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$. We have $\mu_{\mathcal{D}} = \mu$ and $\mathbb{E}[e^{\lambda X}] = e^{\lambda\mu + \frac{\lambda^2\sigma^2}{2}}$, hence $x_{\text{static}} = \frac{\mathcal{W}_0(-e^{-\lambda C-1})+1}{\lambda\mu + \frac{\lambda^2\sigma^2}{2}}$.

Simulations In the experiments in Section 2.5, we randomly sample $\mathcal{D}$ to compute the length of each iteration. For Normal distributions $\text{NORMAL}(\mu, \sigma^2)$, we take $\mu \gg 0$ and sample the distribution until we get a positive value.

2.3.3 First-order approximation

In this section, we show that the first-order approximation (i.e., when the failure rate is very low in front of the distribution parameters) of k_{static} leads to the Young/Daly formula. This result holds for all distributions with finite expectation $\mathbb{E}[e^{\lambda X}]$, hence for all classic distributions. More precisely, we have:

Proposition 2.2. *The first-order approximation k_{FO} of k_{static} obeys the equation*

$$k_{FO} \cdot \mu_{\mathcal{D}} = \sqrt{\frac{2C}{\lambda}}.$$

Proposition 2.2 shows that (the first order approximation of) the average period length of the optimal periodic solution, namely k_{FO} iterations of expected length $\mu_{\mathcal{D}}$, is equal to the Young/Daly period. Note that this result is not surprising but reassuring. Essentially it says that when the inter-arrival time between failure is large in front of the distribution parameters (mean, variance), this distribution can be approximated by a deterministic distribution of size $\mu_{\mathcal{D}}$ to compute the optimal interval size.

Proof. We use Taylor expansions to solve the equation giving the zero of the function $C'_{\text{ind}}(k)$, namely

$$e^{\lambda C}\left(k \log\left(\mathbb{E}[e^{\lambda X}]\right) - 1\right) e^{k \log\left(\mathbb{E}[e^{\lambda X}]\right)} = -1. \tag{2.10}$$

We successively derive that

$$\mathbb{E}[e^{\lambda X}] = 1 + \mathbb{E}[X]\,\lambda + \frac{1}{2}\mathbb{E}\left[X^2\right]\lambda^2 + o\left(\lambda^2\right), \tag{2.11}$$

$$\log \mathbb{E}\left[e^{\lambda X}\right] = \lambda \mathbb{E}\left[X\right] + \frac{\lambda^2}{2}\mathbb{E}\left[X^2\right] - \frac{\left(\lambda \mathbb{E}\left[X\right] + \frac{\lambda^2}{2}\mathbb{E}\left[X^2\right]\right)^2}{2} + o\left(\lambda^2\right)$$
$$= \mathbb{E}\left[X\right]\lambda + \frac{1}{2}\left(\mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2\right)\lambda^2 + o\left(\lambda^2\right), \tag{2.12}$$

$$e^{k \log \mathbb{E}[e^{\lambda X}]} = \left(\mathbb{E}[e^{\lambda X}]\right)^k$$
$$= \left(1 + \lambda \mathbb{E}\left[X\right] + \frac{\lambda^2}{2}\mathbb{E}\left[X^2\right]\right)^k + o\left(\lambda^2\right)$$
$$= 1 + k\mathbb{E}\left[X\right]\lambda + \frac{k}{2}\left(\mathbb{E}\left[X^2\right] + (k-1)\mathbb{E}[X]^2\right)\lambda^2 + o\left(\lambda^2\right). \tag{2.13}$$

By plugging Equations 2.12 and 2.13 in Equation 2.10, we have

$$\left(1+C\lambda+\tfrac{C^2}{2}\lambda^2\right)\left(k\mathbb{E}\left[X\right]\lambda+\tfrac{k\mathbb{E}[X^2]}{2}\lambda^2-\tfrac{k\mathbb{E}[X]^2}{2}\lambda^2-1\right)\left(1+k\mathbb{E}\left[X\right]\lambda+\tfrac{k\mathbb{E}[X^2]}{2}\lambda^2+\tfrac{k(k-1)\mathbb{E}[X]^2}{2}\lambda^2\right)+o\left(\lambda^2\right)=-1.$$

After simplification, we obtain

$$\frac{k^2}{2}\mathbb{E}[X]^2\lambda^2 - C\lambda = o\left(\lambda\right),$$

hence

$$k_{\text{FO}}\mathbb{E}\left[X\right] = \sqrt{\frac{2C}{\lambda}},$$

which corresponds to the Young/Daly formula. □

Simulations In the experiments in Section 2.5, we use

$$k_{\text{FO}} = \max\left(1, \text{round}\left(\frac{1}{\mu_{\mathcal{D}}}\sqrt{\frac{2C}{\lambda}}\right)\right),$$

where round(x) rounds x to the closest integer.

2.4 Dynamic strategies

In Section 2.3, we have studied static solutions where checkpoint locations are decided before the execution. These static decisions are made based upon the distribution $\mathcal{D}$ and the fault rate, but do not depend on the actual length of the iterations in a specific instance of the problem. However, when executing the application, we know on the fly whether some iteration has been much shorter, or much longer, than the average iteration length, and we could take this information into account to decide whether to checkpoint or not. In other words, we take dynamic decisions, at the end of each iteration, and these decisions are based upon the actual work executed since the last checkpoint.

2.4.1 Asymptotic optimality

The dynamic strategy discussed in this section can be stated as follows:

- We fix a threshold W_{th} for the amount of work since the last checkpoint.

- When iteration X_i finishes, if the amount of work since the last checkpoint is greater than W_{th} , then $\delta_i = 1$ (we checkpoint) otherwise $\delta_i = 0$ (we do not checkpoint).

The objective is to determine the value of W_{th} that minimizes the expected execution time of this strategy. However, the expected execution time is much harder to write than for static strategies since the δ_i are now conditional to the values of the X_i. Instead, we make dynamic decisions at the end of each iteration based upon the overhead of the decision (to checkpoint or not). For applications with a large number n of iterations, we minimize the overhead at each step by progressing this way, and always checkpoint the last iteration. This enforces the asymptotic optimality of the strategy when n tends to infinity.

The slowdown H is defined as the ratio

$$H = \frac{\text{actual execution time}}{\text{useful execution time}},$$

so that the slowdown is equal to 1 if there is no cost for fault-tolerance (checkpoints, and re-execution after failures). When an iteration is completed, we compute two values:

- The expected slowdown H_{ckpt} if a checkpoint is taken at the end of this iteration;
- The expected slowdown H_{no} if no checkpoint is taken at the end of this iteration.

The rationale is the following: If $H_{\text{ckpt}} < H_{\text{no}}$, it is better to checkpoint now than waiting for the end of the next iteration, and by induction, than waiting for the end of two or more following iterations. On the contrary, if $H_{\text{no}} < H_{\text{ckpt}}$, it is better not to checkpoint now, in which case we recompute the decision at the end of the next iteration.

We now show how to compute H_{ckpt} and H_{no} . We assume that we just finished an iteration, and that the total amount of work since the last checkpoint (including the last iteration) is w_{dyn}, and write $H_{\text{ckpt}}(w_{\text{dyn}})$ and $H_{\text{no}}(w_{\text{dyn}})$ for the two slowdowns:

Computing H_{ckpt} Recall that Equation 2.1 gives $T_\lambda(W, C, D, R)$, the expected execution time to perform a work of size W followed by a checkpoint of size C, with downtime D and recovery R. The expected time to compute w_{dyn} was $T(w_{\text{dyn}}, 0, D, R)$, and the expected time to checkpoint now is $T(0, C, D, R + w_{\text{dyn}})$: this is because if a failure strikes during the checkpoint, we have to reexecute w_{dyn}. Finally, the useful execution time is w_{dyn}, hence

$$\begin{aligned} H_{\text{ckpt}}(w_{\text{dyn}}) &= \frac{T(w_{\text{dyn}}, 0, D, R) + T(0, C, D, R + w_{\text{dyn}})}{w_{\text{dyn}}} \\ &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\frac{(e^{\lambda w_{\text{dyn}}} - 1) + e^{\lambda w_{\text{dyn}}}\left(e^{\lambda C} - 1\right)}{w_{\text{dyn}}} \\ &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\frac{e^{\lambda\left(w_{\text{dyn}}+C\right)} - 1}{w_{\text{dyn}}}. \end{aligned} \tag{2.14}$$

Computing H_{no} If we do not checkpoint now but only at the end of the next iteration of length $X = w$ (drawn from distribution $\mathcal{D}$), the actual execution time will be $T(w_{\text{dyn}}, 0, D, R) + T(w, C, D, R +$

w_{dyn}) and the useful time will be $w_{\text{dyn}} + w$. Hence we need to take the expectation of the ratio and obtain

$$\begin{aligned} H_{\text{no}}(w_{\text{dyn}}) &= \mathbb{E}\left(\frac{T(w_{\text{dyn}}, 0, D, R) + T(X, C, D, R + w_{\text{dyn}})}{w_{\text{dyn}} + X}\right) \\ &= \int_D \frac{T(w_{\text{dyn}}, 0, D, R) + T(w, C, D, R + w_{\text{dyn}})}{w_{\text{dyn}} + w} f(w)dw, \end{aligned}$$

where $f(x)$ is the PDF of $\mathcal{D}$ and $\mathcal{D}$ is its domain. Computing the expectation of this ratio is too difficult, and we approximate it by taking the ratio of the expectations (actual time over useful time), so we redefine H_{no} by

$$\begin{aligned} H_{\text{no}}(w_{\text{dyn}}) &= \frac{\mathbb{E}\left[T(w_{\text{dyn}}, 0, D, R) + T(X, C, D, R + w_{\text{dyn}})\right]}{\mathbb{E}\left[w_{\text{dyn}} + X\right]} \\ &= \frac{T(w_{\text{dyn}}, 0, D, R) + \mathbb{E}\left[T(X, C, D, R + w_{\text{dyn}})\right]}{w_{\text{dyn}} + \mathbb{E}[X]} \\ &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\frac{(e^{\lambda w_{\text{dyn}}} - 1) + e^{\lambda w_{\text{dyn}}}\left(e^{\lambda C}\mathbb{E}[e^{\lambda X}] - 1\right)}{w_{\text{dyn}} + \mathbb{E}[X]} \\ &= e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\frac{e^{\lambda\left(w_{\text{dyn}} + C\right)}\mathbb{E}[e^{\lambda X}] - 1}{w_{\text{dyn}} + \mathbb{E}[X]}. \end{aligned} \tag{2.15}$$

The last line of Equation 2.15 is obtained using Equation 2.5.

Computing W_{th} By defintion, W_{th} is the threshold value where

$$H_{\text{ckpt}}(W_{\text{th}}) = H_{\text{no}}(W_{\text{th}}).$$

Using Equations 2.14 and 2.15, we obtain

$$W_{\text{th}}\left(e^{\lambda(W_{\text{th}} + C)}\mathbb{E}[e^{\lambda X}] - 1\right) = (W_{\text{th}} + \mathbb{E}[X])\left(e^{\lambda(W_{\text{th}} + C)} - 1\right). \tag{2.16}$$

After simplification, we have

$$\left(\left(\mathbb{E}[e^{\lambda X}] - 1\right)W_{\text{th}} - \mathbb{E}[X]\right)e^{\lambda(W_{\text{th}} + C)} = -\mathbb{E}[X],$$

by multipling $\frac{\lambda}{\mathbb{E}[e^{\lambda X}] - 1}e^{-\lambda\left(C + \frac{\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}\right)}$ on both sides of the equation, we have

$$te^t = -\frac{\lambda\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}e^{-\lambda\left(C + \frac{\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}\right)},$$

where $t = \lambda W_{\text{th}} - \frac{\lambda\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}$. Finally, we derive the threshold value:

$$W_{\text{th}} = \frac{1}{\lambda}\mathcal{W}_0\left(-\frac{\lambda\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}e^{-\lambda\left(C + \frac{\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}\right)}\right) + \frac{\mathbb{E}[X]}{\mathbb{E}[e^{\lambda X}] - 1}.$$

2.4.2 First-order approximation

In this section, we show that the first-order approximation of W_{th} leads to the Young/Daly formula. This result holds for all distributions with finite expectation $\mathbb{E}[e^{\lambda X}]$, hence for all classic distributions. More precisely, we have:

Proposition 2.3. *The first-order approximation W_{FO} of W_{th} obeys the equation*

$$W_{FO} = \sqrt{\frac{2C}{\lambda}}. \tag{2.17}$$

Equation 2.17 shows that the first order approximation of the threshold value W_{th} , namely W_{FO}, is equal to the Young/Daly period.

Proof. We use Taylor expansions to solve Equation 2.16. After simplification, we have

$$W_{\text{th}} e^{\lambda(W_{\text{th}}+C)} \left(\mathbb{E}\left[e^{\lambda X}\right] - 1\right) = \mathbb{E}\left[X\right] \left(e^{\lambda(W_{\text{th}}+C)} - 1\right), \tag{2.18}$$

by plugging Equation 2.11 in Equation 2.18, we have

$$W_{\text{th}} \left(1 + \lambda\left(W_{\text{th}} + C\right) + \frac{\lambda^2 (W_{\text{th}} + C)^2}{2}\right) \left(\lambda \mathbb{E}\left[X\right] + \frac{\lambda^2 \mathbb{E}\left[X^2\right]}{2}\right) = \mathbb{E}\left[X\right] \left(\lambda\left(W_{\text{th}} + C\right) + \frac{\lambda^2 (W_{\text{th}} + C)^2}{2}\right).$$

After simplification, we obtain

$$\frac{\lambda W_{\text{th}}^2}{2} - C\lambda = o\left(\lambda\right),$$

hence

$$W_{\text{FO}} = \sqrt{\frac{2C}{\lambda}},$$

which corresponds to the Young/Daly formula. □

Simulations In the experiments in Section 2.5, we use Equation 2.17.

2.5 Experiments

In this section, we describe the experiments conducted to assess the efficiency of static and dynamic solutions, as well as the accuracy of the Young/Daly formula. Propositions 2.2 and 2.3 show that when the number of failures is low, the Young/Daly formula is a good approximation. We aim at showing experimentally that this remains the case with higher failure rates, when the first-order approximation is no longer valid. In Section 2.5.1, we detail the experimental methodology with all simulation parameters. Results are presented in Section 2.5.2.

2.5.1 Experimental methodology

For each experiment, the evaluations are performed on 10,000 randomly generated instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_{10000}\}$. For all i, an instance $\mathcal{I}_i$ is a pair $(\mathcal{S}_i, \mathcal{F}_i)$, where $\mathcal{S}_i$ (resp. $\mathcal{F}_i$) is the application (resp. failure) scenario associated with the instance.

The algorithms are implemented in MATLAB and R. The corresponding code is available at [34]. This simulator computes the makespan for our static strategy, the Young/Daly-static strategy, our dynamic strategy, and the Young/Daly-dynamic strategy.

Application scenarios We consider an iterative application composed of $n = 1,000$ consecutive iterations. We assume that the execution time of each iteration follows a probability distribution $\mathcal{D}$, where $\mathcal{D}$ is either $\text{UNIFORM}(a, b)$, $\text{GAMMA}(\alpha, \beta)$ or truncated $\text{NORMAL}(\mu, \sigma^2, [0, \infty))$ (see Section 2.3.2 for the corresponding PDFs). The default instantiations for these distributions are $\mu_{\mathcal{D}} = 50$ with $\text{UNIFORM}[20, 80]$, $\text{GAMMA}(25, 0.5)$ and $\text{NORMAL}(50, 2.5^2)$ (recall that we sample the latter one until a positive value is found). We also study the impact of the standard deviation σ.

Failure scenarios We consider different failure rates. To allow for consistent comparisons of results across different iterative processes with different probability distributions, we fix the probability that failure occurs during each iteration, which we denote at p_{fail}, and then simulate the corresponding failure rate. Formally, for a given p_{fail} value, we compute the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda(\mu_{\mathcal{D}}+C)}$, where $\mu_{\mathcal{D}} + C$ is the average length of an iteration followed by a checkpoint. We conduct experiments for seven p_{fail} values: $10^{-3}, 10^{-2.5}, 10^{-2}, 10^{-1.5}, 10^{-1}, 10^{-0.5}$ and $10^{-0.1}$. For example, $p_{\text{fail}} = 10^{-2}$ means one failure may occur every 100 iterations.

Checkpointing costs Important factors that influence the performance of checkpointing strategies are the checkpointing and recovery costs. We set checkpoint time as $C = \eta\mu_{\mathcal{D}}$, where η is the proportion of checkpoint time to the expectation of iteration time. And we set recovery time as $R = C$ and fixed downtime as $D = 1$. We conducted the experiments with $\eta = 0.1$.

Static strategies The algorithm used to simulate the static strategies is detailed in Algorithm 1.

Algorithm 1: Algorithm to choose optimal k (checkpoint after k iterations)

Input: Total number of iterations n, Execution time of each iteration X_i, Failure time F_m, Checkpoint time C, Recovery time R, Downtime D.
Output: The makespan t

while $i \leq n$ **do**
  if $t + X_i + \ldots + X_{i+k-1} + C \leq F_m$ **then**
    /* success between ith iteration and $i+k-1$th iteration */
    $t \leftarrow t + X_i + \ldots + X_{i+k-1} + C$
    $i \leftarrow i + k$
  else
    /* failure bewteen ith iteration and $i+k-1$th iteration */
    if $F_m + D + R \leq F_{m+1}$ **then**
      /* no failure in recovery */
      $t \leftarrow F_m + D + R$
      $m \leftarrow m + 1$
    else
      /* failure in recovery */
      $t \leftarrow F_{m+1}$
      $m \leftarrow m + 2$
      while $t + D + R > F_m$ **do**
        /* look for first successful recovery */
        $t \leftarrow F_m$
        $m \leftarrow m + 1$
      $t \leftarrow t + D + R$
return t

For an instance $\mathcal{I}$, we define $MS_{\text{sim\_sta}}(k)(\mathcal{I})$ to be the makespan when checkpointing every k iterations. In addition, we define the average value

$$\overline{MS}_{\text{sim\_sta}}(k) = \frac{1}{10000} \sum_{i=1}^{10000} MS_{\text{sim\_sta}}(k)(\mathcal{I}_i)$$

and the minimal average makespan over k

$$MS^{\min}_{\text{sim\_sta}} = \min_k \overline{MS}_{\text{sim\_sta}}(k).$$

This minimum is reached for $k = k_{\text{sim}}$.

We also compare the simulations with the theoretical model. We use $\mathbb{E}[MS_{\mathcal{D}}](k) = n \cdot e^{\lambda R}\left(\frac{1}{\lambda} + D\right) C_{ind}(k)$, where C_{ind} depends on $\mathcal{D}$, and $n = 1,000$. We define $MS^{\text{OPT}}_{\text{the\_sta}} = \mathbb{E}[MS_{\mathcal{D}}](k_{\text{static}})$. Finally, we define the Young/Daly static as $MS_{\text{YD\_sta}} = MS_{\text{sim\_sta}}(k_{\text{FO}})$ and $\overline{MS}_{\text{YD\_sta}}$ as its average value over all instances.

Dynamic strategies The algorithm used to simulate the dynamic strategies is detailed in Algorithm 2.

Algorithm 2: Algorithm to choose optimal w_{dyn} (checkpoint when the amount of work since last checkpoint is greater than w_{dyn})

Input: Total number of iterations n, Execution time of each iteration X_i, Failure time F_m, Checkpoint time C, Recovery time R, Downtime D, threshold w_{dyn}.
Output: The running time t

while $i \leq n$ **do**
  for $k = 1, k \leq n - i + 1, k++$ **do**
    if $X_i + \ldots + X_{i+k-1} \geq w_{dyn}$ **then**
      $k_{\text{dyn}} \leftarrow k$
      break
  if $t + X_i + \ldots + X_{i+k_{dyn}-1} + C \leq F_m$ **then**
    /* success between ith iteration and $i + k_{\text{dyn}} - 1$th iteration */
    $t \leftarrow t + X_i + \ldots + X_{i+k_{\text{dyn}}-1} + C$
    $i \leftarrow i + k_{\text{dyn}}$
  else
    /* failure between ith iteration and $i + k_{\text{dyn}} - 1$th iteration */
    if $F_m + D + R \leq F_{m+1}$ **then**
      /* no failure in recovery */
      $t \leftarrow F_m + D + R$
      $m \leftarrow m + 1$
    else
      /* failure in recovery */
      $t \leftarrow F_{m+1}$
      $m \leftarrow m + 2$
      while $t + D + R > F_m$ **do**
        /* look for first successful recovery */
        $t \leftarrow F_m$
        $m \leftarrow m + 1$
      $t \leftarrow t + D + R$
return t

We simulate the dynamic strategy with different threshold values $W = \gamma \cdot W_{\text{th}}$ with $\gamma \in \{0.1, 0.2, \ldots, 2\}$. For an instance $\mathcal{I}$, we define $MS_{\text{sim\_dyn}}(W)(\mathcal{I})$ as the makespan with threshold W, and $\overline{MS}_{\text{sim\_dyn}}(W)$ as its average value over all instances. Then we let $MS^{\text{OPT}}_{\text{sim\_dyn}} =$

$\min_W \overline{MS}_{\text{sim\_dyn}}(W)$. It is reached for $W = W_{\text{sim}}$. Finally, we define the Young/Daly dynamic as $MS_{\text{YD\_dyn}} = MS_{\text{sim\_dyn}}(W_{\text{FO}})$ and $\overline{MS}_{\text{YD\_dyn}}$ as its average value over all instances.

2.5.2 Results

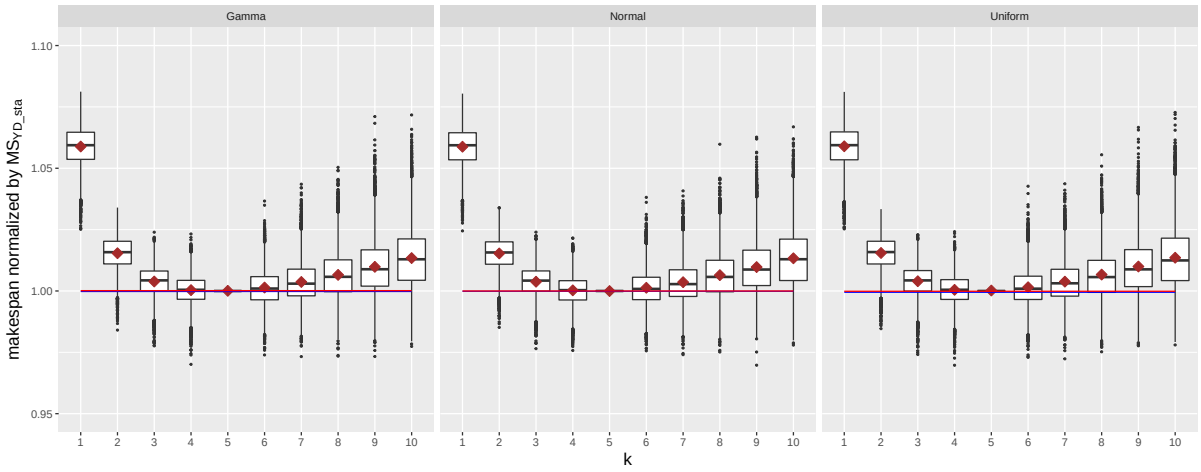


Figure 2.1: Boxplots represent the performance of the static strategy that chooses k. Brown diamonds plot $\mathbb{E}[MS_{\mathcal{D}}](k)$ (theoretical makespan). The blue (resp. red) line represents the makespan obtained by the optimal dynamic strategy $\overline{MS}_{\text{sim\_dyn}}(W_{\text{th}})$ (resp. the YD-dynamic strategy $\overline{MS}_{\text{YD\_dyn}}$).

Table I: Simulation for static case.

$p_{\text{fail}} = 10^{-2}$	Gamma	Normal	Uniform
k_{sim}	5	5	5
x_{static}	4.6114	4.6122	4.6097
k_{static}	5	5	5
$\frac{1}{\mu_{\mathcal{D}}}\sqrt{\frac{2C}{\lambda}}$	4.6787	4.6787	4.6787
k_{FO}	5	5	5

Static strategies The results from the static case are reported in Figure 2.1. Specifically, the box plots represent the evolution of the function $\mathcal{I} \mapsto \frac{MS_{\text{sim\_sta}}(k)}{MS_{\text{YD\_sta}}}(\mathcal{I})$ (for different values of k), and the black lines correspond to its mean. The diamonds represent the average: $\frac{1}{10000}\sum_{i=1}^{10000} \frac{\mathbb{E}[MS_{\mathcal{D}}](k)}{MS_{\text{YD\_sta}}(\mathcal{I}_i)}$.

The first important result from this plot is the experimental validation of our model. Indeed, the blacklines and diamonds are almost identical for all k. The closer we get to the optimal value k_{static}, the closer the theoretical makespan gets to the simulation makespan. In particular, for $k = 5$, which corresponds to k_{FO} (and k_{static}), the makespan obtained is exactly the same for $MS_{\text{sim\_sta}}$ and $MS_{\text{YD\_sta}}$, leading to a ratio of 1 in all cases: the boxplot contains a single value. This is in accordance with Table I. From Table I, it can be observed that $k_{\text{sim}} = k_{\text{static}} = k_{\text{FO}}$.

A consequence is that the solution k_{static} (as well as Young/Daly's solution) always provides the optimal expected makespan, in coherence with the theoretical results. Because it is a stochastic process, it can not always give the optimal makespan, but we see from these figures that it is always within 3% of the makespan obtained by other strategies, which shows the robustness of this choice.

As expected the ratio $MS^{\text{OPT}}_{\text{the\_sta}}/MS_{\text{YD\_sta}}$ is equal to 1 since in those cases $k = 5$ for k_{static} and k_{FO}. We have tried a large range of values to check if there are cases when they are not and have found that they almost always are (see Figures 2.3 and 2.4 and comments).

In order to compare static strategies with dynamic strategies, we plot blue and red lines corresponding to the ratios $\overline{MS}_{\text{sim\_dyn}}(W_{\text{th}})/\overline{MS}_{\text{YD\_sta}}$, and $\overline{MS}_{\text{YD\_dyn}}/\overline{MS}_{\text{YD\_sta}}$, respectively. Both lines are very close to 1, meaning that these two dynamic strategies have the same performance as the optimal static strategy.

Overall the conclusions of this section is that the simple strategy based on the Young/Daly setting remains a good and robust solution for stochastic applications, and can safely be used in this context.

Table II: Simulation for dynamic case.

$p_{\text{fail}} = 10^{-2}$	Gamma	Normal	Uniform
γ	1.0	1.0	1.0
W_{sim}	206.0492	206.8876	204.2743
$MS^{\text{OPT}}_{\text{sim\_dyn}}$	52267	52264	52267
W_{th}	206.0492	206.8876	204.2743
$\overline{MS}_{\text{sim\_dyn}}(W_{\text{th}})$	52267	52264	52267
W_{FO}	233.9328	233.9328	233.9328
$\overline{MS}_{\text{YD\_dyn}}$	52284	52271	52288

Dynamic strategies We compare our dynamic strategy with the threshold obtained with the Young/-Daly formula. For each γ, we report the makespan of 10,000 random simulations using boxplots. From Table II, it can be observed that $W_{\text{sim}} = W_{\text{th}}$. Of course, giving more precision to γ may give slightly better performance, but the gain remains negligible. Contrarily to the static case, W_{th} and W_{FO} are different (up to 15%). However the performance obtained (Figure 2.2) are similar, and again the Young/Daly formula seems a safe bet given its simplicity of use.

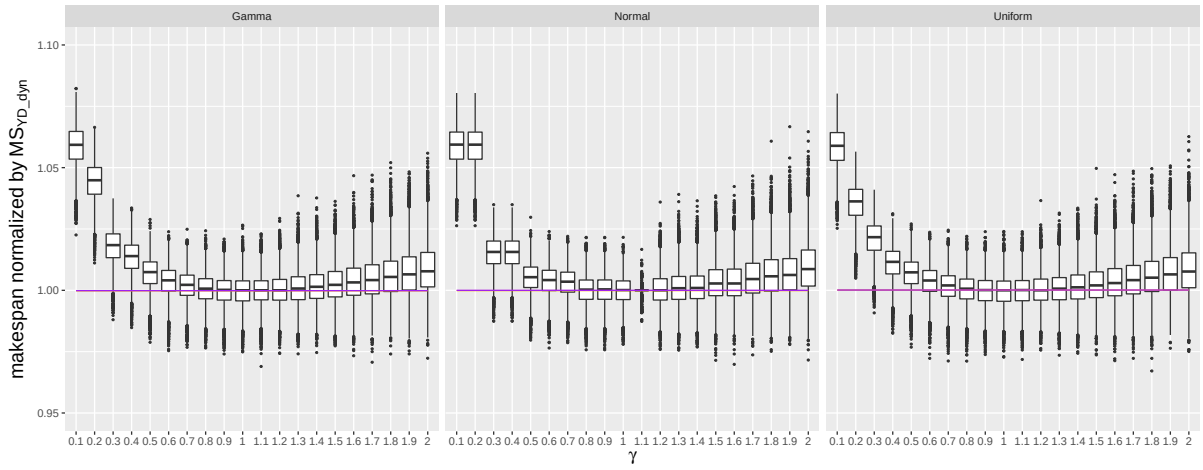


Figure 2.2: Boxplots represent the performance of the dynamic strategy that chooses a threshold of $\gamma \cdot W_{\text{th}}$. The orange (resp. purple) line represents the makespan obtained by the optimal static strategy $\overline{MS}_{\text{sim\_sta}}(k_{\text{static}})$ (resp. the YD-static strategy $\overline{MS}_{\text{YD\_sta}}$).

We plot orange and purple lines corresponding to the ratios $\overline{MS}_{\text{sim\_sta}}(k_{\text{static}})/\overline{MS}_{\text{YD\_dyn}}$ and $\overline{MS}_{\text{YD\_sta}}/\overline{MS}_{\text{YD\_dyn}}$, respectively. As in Figure 2.1, these ratios are very close to 1, meaning that the static and the dynamic strategies give similar results both when using optimal parameters or the one approximated using the Young/Daly formulas.

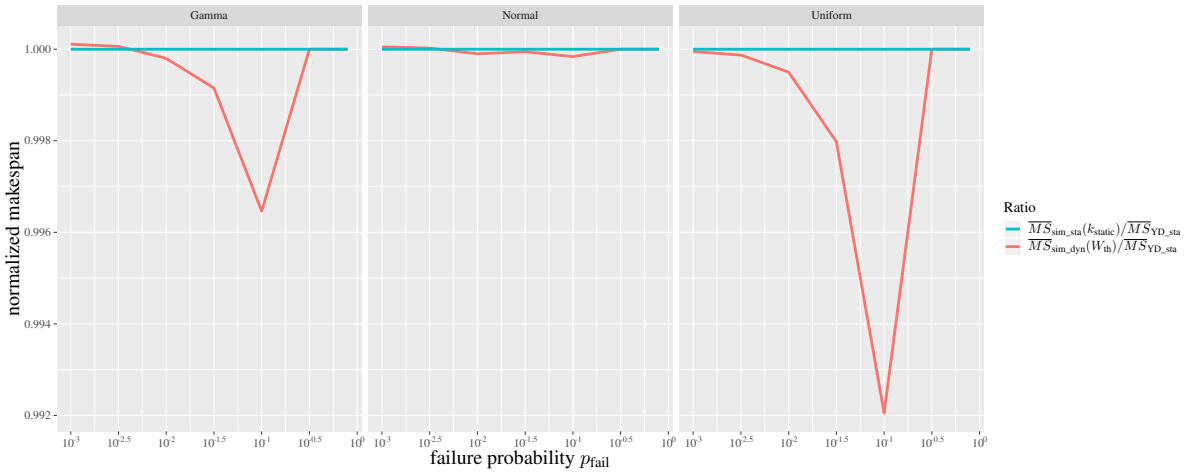


Figure 2.3: Simulation with varying failure probability

Both strategies for varying p_{fail} In order to study the sensibility of our results to the failure probability, we compare in Figure 2.3 the makespan obtained by the static Young/Daly approximation ($\overline{MS}_{\text{YD\_sta}}$) to the makespan obtained by the simulation when using the optimal k_{static} ($\overline{MS}_{\text{sim\_sta}}(k_{\text{static}})$), and the one of the optimal dynamic strategy ($\overline{MS}_{\text{sim\_dyn}}(W_{\text{th}})$). We observe that the first two makespans are always equal, because in all cases $k_{\text{FO}} = k_{\text{static}}$. The optimal dynamic strategy is sometimes slightly better that the static ones, but with a gap smaller than 0.5% for all failure probabilities.

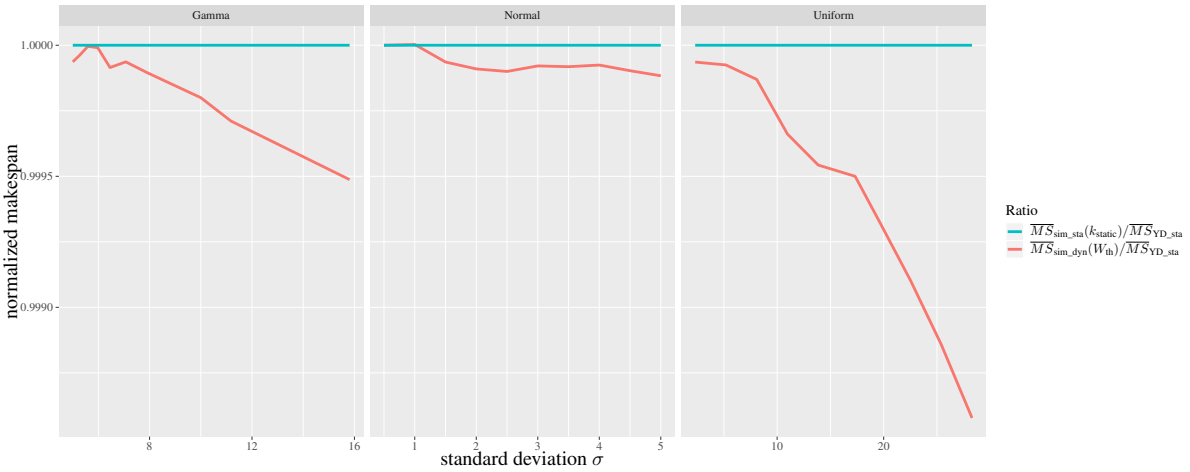


Figure 2.4: Simulation with varying standard deviation

Both strategies for varying σ We vary the standard deviation σ of each distribution of execution times in Figure 2.4. Again, there is no difference between k_{static} and k_{FO} in all tested cases, leading to a ratio $\overline{MS}_{\text{sim\_sta}}(k_{\text{static}})/\overline{MS}_{\text{YD\_sta}}$ constant and equal to 1. The optimal dynamic strategy is again very close, with a gap smaller than 0.05% even for very large deviations.

Both strategies for varying η Finally, we vary the proportion η of checkpoint time to expected iteration time for Gamma distributions in Figure 2.5. As expected, both the optimal k and W increase together with checkpoint time. The optimal static and dynamic strategies are still very close, with a gap larger or smaller than 0.05%, even for very large checkpoint times.

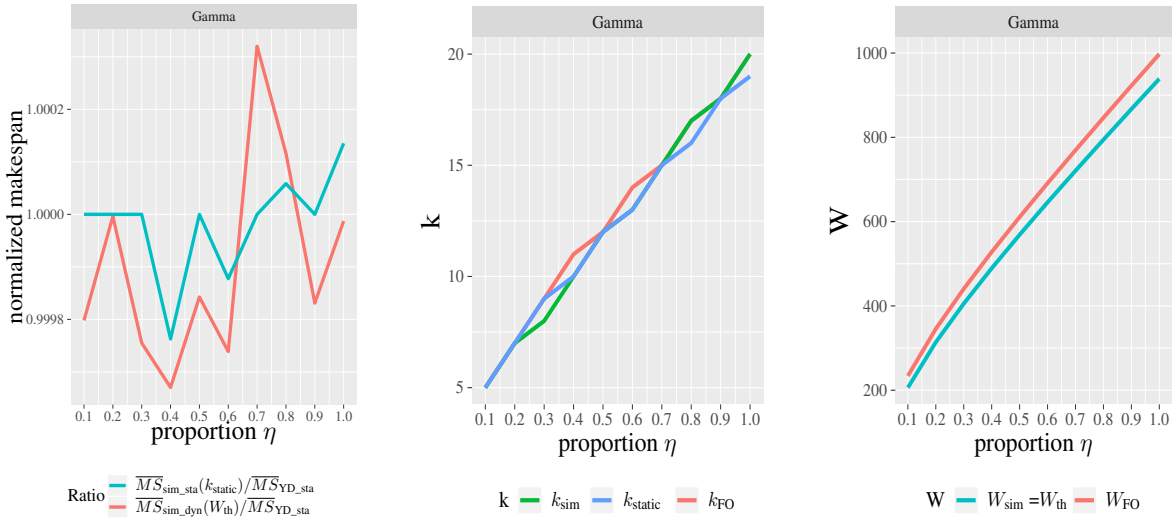


Figure 2.5: Simulation with varying the proportion of checkpoint time η to the expected iteration time.

2.6 Conclusion

We have introduced and analyzed checkpointing strategies for iterative applications. The key novelty is that this work does not assume deterministic iterations, but instead models execution times with probabilistic distributions. Our first main contribution is to provide a closed-form formula, valid for any distribution, to compute the optimal period at which one should checkpoint as a function of the failure rate. Then, we provide efficient solutions for non periodic, online solutions, where one decides on the fly whether to perform a checkpoint or to perform an additional iteration. In addition to these solutions, we study the behavior of the Young/Daly solution. We then show the following: as a first-order approximation, both periodic and non periodic solutions converge to the Young/Daly formula. All these derivations are quite technical, and constitute a major extension of the deterministic case.

In addition, we are able to show via extensive simulations that the Young/Daly formula is in general an excellent solution for non-deterministic execution times. This is done in two steps: (i) we show that our mathematical model is extremely accurate, since the mathematical formula fits almost perfectly the evaluated execution time; and (ii) the performance of the Young/Daly formula is always within one percent that of the optimal strategy that we obtained.

This study opens interesting problems in this area, such as extending this study to multi-level checkpointing protocols, which correspond to state-of-the-art approaches but are already quite difficult to

model and optimize analytically in a deterministic setting. Extending known results to a stochastic framework is a challenging problem.

Chapter 3

Deterministic iterative applications

3.1 Introduction

As mentioned in the introduction, we proceed to deterministic iterative applications in this chapter. We focus on designing optimal checkpoint strategies for iterative workflows expressed as *pipelined linear workflows*: we consider workflows made of a large number of iterations, each iteration being a linear chain of parallel tasks. The typical example is an application consisting of an outer loop "`While convergence is not met, do`", and where the loop body includes a sequence of large parallel operations. The objective is to find which task outputs should be saved on stable storage in order to minimize the expected duration of the whole computation. To the best of our knowledge, this is an open problem, despite the practical importance and ubiquity of pipelined linear workflows in High-Performance Computing (HPC). Indeed, the simple case of a unique linear chain of tasks (a pipelined linear workflow with a single iteration) has been solved by Toueg et al. [105] using a dynamic programming algorithm. On the contrary, the problem for workflows with general directed graphs has been shown #P-complete[1] [50]. The study for pipelined linear workflows (a linear chain with several iterations) is challenging, and the main contribution of this chapter is to provide a complete answer:

- We prove that there exists an optimal checkpointing strategy which is periodic. It consists in a pattern of task outputs to checkpoint, where this pattern spans over a set of iterations of bounded size. This pattern is repeated over and over throughout the execution.
- We provide a dynamic programming algorithm which is polynomial in the number of operations included in the outer loop to compute the optimal periodic checkpoint pattern. The complexity of the algorithm does not depend on the number of iterations of the outer loop. This pattern may well checkpoint many different tasks, and this across many different iterations.
- We conduct an extensive set of simulations to compare the optimal strategy to four natural competitor strategies: the first checkpoints after each task, the second after each iteration, while the last two are extensions of the Young/Daly formula for iterative applications. Our simulations with both synthetic and real-life workflow instances demonstrate that our optimal strategy provide improvement over the simpler competitors.

The rest of the chapter is organized as follows. Section 3.2 presents a detailed model for the problem and states the objective function. Section 3.3 outlines the optimal checkpoint strategy. Section 3.4

[1] #P-complete problems are counting problems that are at least as hard as NP-complete problems

reports a comprehensive set of experimental results, based upon both synthetic workflows and on workflows arising from two real-life applications. Finally, Section 3.5 provides concluding remarks and directions for future work.

3.2 Model

In this section, we detail the application and platform models. Then we define checkpoint strategies and formally state the optimization objective. See Table I for a summary of main notations.

Application	
n	number of tasks per iteration
N_{iter}	number of iterations
a_i	task number i, $0 \leq i < n$, in each iteration
t_i	duration of task a_i
c_i, r_i	checkpoint and recovery time for task a_i
T	length of an iteration
Platform	
D	downtime
$\mu = 1/\lambda$	platform MTBF (λ is the parameter of the failure distribution)
Schedule $\mathcal{S}$	
m_i	task number m_i is checkpoint number i in the schedule
$C_i^{\mathcal{S}}$	checkpoint cost at end of chunk number i
$W_i^{\mathcal{S}}$	length of chunk number i (including tasks number $m_{i-1}+1$ to m_i)
$R_{i-1}^{\mathcal{S}}$	recovery cost when re-executing chunk number i
$\mathrm{SD}(\mathcal{S})$	slowdown of schedule $\mathcal{S}$
P	checkpoint path or pattern
$\ell(P)$	length of checkpoint path P
$\mathcal{C}(P)$	expected execution time, or cost, of checkpoint path P
$\mathrm{SD}(P)$	slowdown of a path P
w_{c_i}	Young/Daly period for checkpoint type c_i, where $w_{c_i} = \sqrt{\frac{2c_i}{\lambda}}$
$2nM^\star$	upper bound for length of optimal period, where $M^\star = \max_i w_{c_i} + T$
$k^\star$	number of iterations $\lceil \frac{M^\star}{T} \rceil$ taking place during time $M^\star$

Table I: Summary of main notations.

3.2.1 Application model

We consider an iterative application $\mathcal{A}$. Each iteration of the application consists of n parallel tasks a_i, where $0 \leq i < n$, task a_i has length t_i and memory footprint M_i. We define the length of an iteration as $T = \sum_{i=0}^{n-1} t_i$.

The tasks are executed consecutively: let $i[n]$ denote the remainder of the integer division of i by n (modulo operation); then a task a_i is always followed by a task $a_{i+1[n]}$. We assume that the application executes for a long time and consider an unbounded number of iterations (but we use $1,000$ iterations in the experiments). For short we write $\mathcal{A} = (a_0, \ldots, a_{n-1})^\infty$. As stated before, we execute tasks one after the other. The first executed task is a_0, followed by a_1, and so on. The n^{th} task is a_{n-1} and the

$(n+1)$st task is a_0 again. In general, the k^{th} task is $a_{k-1[n]}$. Note that we index tasks from 0 to use the modulo operation, hence this shift when counting executed tasks.

We assume that the tasks of the application can be checkpointed at the end of their execution. We consider a general model where the checkpoint time of task a_i is c_i and its recovery time is r_i. We refer to c_i and r_i as operations of type i. We do not assume that $c_i = r_i$; instead, we simply assume monotone I/O costs:

$$\text{for all } i, j, \quad c_i \geq c_j \implies r_i \geq r_j.$$

Essentially this assumption states that if a task is longer to checkpoint than another one, then restarting from this checkpoint is also longer. This is coherent with the fact that checkpoint and recovery costs are often closely related, and are a function of the volume of data to save. Furthermore, this assumption is general enough to account for different read and write bandwidths.

We assume that all task parameters (execution time, checkpoint, recovery) are known. This is a natural assumption for iterative applications which repeat each task a large number of times and can determine their characteristics either through an analytical model or by repetitive sampling. However, to assess the robustness of the approach, we also report experiments using stochastic execution times derived from a Normal probability distribution.

3.2.2 Platform model

We consider a parallel platform whose nodes are subject to fail-stop errors, or failures. A failure, such as a node crash, interrupts the execution of the node and provokes the loss of its whole memory. 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 consists of using a checkpoint-restart mechanism: the state of the application is periodically checkpointed, which means that all participating nodes take a checkpoint simultaneously: this is the standard coordinated checkpointing protocol which is routinely used on large-scale platforms [23], where each node 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 which will replace the faulty processor [29, 55]. 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). Then the execution is resumed from that point on, rather than starting again from scratch.

We assume that the iterative application 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 MTBF is $\mu = \frac{1}{\lambda}$ and corresponds to the MTBF of individual processors divided by the total number of processors enrolled in the application [55]. As stated in the introduction, 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 [20, 39]. Hence, a parallel applications using a significant fraction of the platform will typically experience a failure every few days.

The key for an efficient checkpointing policy is to decide how often to checkpoint. Young [110] and Daly [29] derived the well-known Young/Daly formula $\mathcal{P}_{YD} = \sqrt{2\mu C}$ for the optimal checkpointing period, where μ is the application MTBF and C is the checkpoint duration, as defined above.

3.2.3 Schedule

Informally, a schedule defines which tasks are checkpointed. A priori, there is no reason for a schedule to enforce a regular pattern of checkpoints that repeats over time. In other words, a schedule can be aperiodic. However, one major contribution of this work is to show that periodic schedules are optimal, and to exhibit the optimal period as the output of a polynomial-time algorithm. We need a few definitions before stating the objective function to be minimized by *optimal* schedules. First we identify a schedule with the list of the tasks that it checkpoints:

Definition 3.1 (Schedule)**.** *A schedule $\mathcal{S}$ is an infinite increasing sequence $\mathcal{S} = (m_1, m_2, \dots)$ which represents the list of checkpointed tasks: the m_i^{th} task (i.e., task number m_i) is checkpointed, and the tasks whose number does not belong to the list are not checkpointed.*

In other words, checkpoint number i in the schedule takes place at the end of task number m_i. The cost to checkpoint that task m_i is $c_{m_i-1[n]}$ (because of the index shift noted above). Without loss of generality, we assume that the schedule checkpoints infinitely many tasks, i.e., $\lim_{i\to\infty} m_i = \infty$. Indeed, consider any task in the application: eventually there must be a checkpoint after that task, otherwise the expected execution time from that task is not bounded, because the fault-rate λ is nonzero.

A schedule $\mathcal{S}$ can be viewed as a succession of task chunks between two consecutive checkpoints. We use the following notations for the i^{th} chunk between checkpoint number $i-1$ (or the beginning of the execution if $i = 1$) and checkpoint number i:

- The length of the tasks in the chunk is
$$W_i^{\mathcal{S}} = \sum_{j=m_{i-1}+1}^{m_i} t_{j-1[n]}.$$
- The checkpoint cost at the end of the chunk is the cost of checkpoint number i, namely
$$C_i^{\mathcal{S}} = c_{m_i-1[n]}.$$
- The recovery cost when re-executing the chunk is the cost of recovering from checkpoint number $i-1$, namely
$$R_{i-1}^{\mathcal{S}} = r_{m_{i-1}-1[n]}.$$
When $i = 1$ (first chunk), we let $m_0 = 0$, and $R_0^{\mathcal{S}}$ denotes the cost of reading input data.

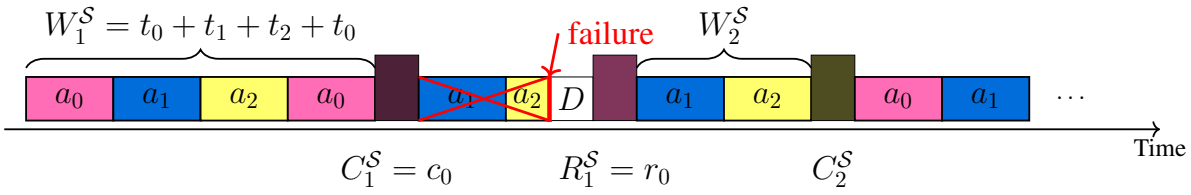


Figure 3.1: Notations drawn in a schedule for an application with $n = 3$ tasks.

3.2.4 Objective function

Intuitively, a good schedule will minimize the slowdown during the execution. This slowdown comes from two sources of overhead: the checkpoints that are inserted, and the time lost due to failures. When a

failure strikes during execution, the work executed since the last checkpoint is lost; there is a downtime, followed by a recovery, and then the re-execution of the work that has been lost due to the failure. Altogether, the overhead is not deterministic and varies from one execution to the other, hence we aim at minimizing the slowdown in expectation.

Given a schedule $\mathcal{S}$, we rely on a well-known formula to compute the expected execution time of a chunk. Indeed, the expected execution time $\mathbb{E}_\lambda(w, c, r)$ to execute w consecutive seconds of work followed by a checkpoint of size c with a recovery of size r is given by [55]

$$\mathbb{E}_\lambda(w, c, r) = \left(\frac{1}{\lambda} + D\right) e^{\lambda r} \left(e^{\lambda(w+c)} - 1\right).$$

The expected time to execute the chunk number i is thus $\mathbb{E}_\lambda(W_i^{\mathcal{S}}, C_i^{\mathcal{S}}, R_{i-1}^{\mathcal{S}})$. Hence the expected time to execute the first i chunks is $\sum_{j=1}^{i} \mathbb{E}_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})$. The slowdown incurred for the first i chunks (i.e., up to checkpoint number i) is therefore

$$\mathrm{SD}_i(\mathcal{S}) = \frac{\sum_{j=1}^{i} \mathbb{E}_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})}{\sum_{j=1}^{m_i} t_{j-1[n]}}. \tag{3.1}$$

In Equation 3.1, the numerator is the expected time to execute the first i chunks, while the denominator is the duration of the tasks up to checkpoint number i, which corresponds to the resilience-free and failure-free execution.

Unfortunately, there is no reason that $\lim_{i\to\infty} \mathrm{SD}_i(\mathcal{S})$ would exist. However, we can use the upper limit of $\mathrm{SD}_i(\mathcal{S})$ to define the slowdown of schedule $\mathcal{S}$:

Definition 3.2 (Slowdown)**.** *The slowdown SD of a schedule $\mathcal{S}$ is:*

$$SD(\mathcal{S}) = \overline{\lim_{i\to\infty}} SD_i(\mathcal{S}).$$

We know that this upper limit is bounded for some schedules. Consider for instance the schedule $\mathcal{S}$ that checkpoints all tasks: $m_i = i$ for all $i \geq 1$. This schedule repeats the same pattern of checkpoints every iteration, so that its slowdown is

$$\mathrm{SD}(\mathcal{S}) = \frac{\mathrm{SD}_n(\mathcal{S})}{T} = \frac{\sum_{j=0}^{n-1} \left(\frac{1}{\lambda} + D\right) e^{\lambda r_{j-1[n]}} \left(e^{\lambda(t_j+c_j)} - 1\right)}{T}.$$

Recall that T is the length of an iteration. We are now ready to define an optimal schedule:

Definition 3.3 (Optimal schedule)**.** *A schedule is optimal if its slowdown $SD(\mathcal{S})$ is minimal over all possible schedules.*

Note that the definition does not assume that there exists a unique optimal schedule. A major contribution of this chapter is to show that there exists an optimal schedule which is periodic, i.e., which repeats the same pattern of checkpoints after some point (see below for the formal definition). This important result will allow us to consider only a finite number of candidate schedules, and to design a polynomial-time algorithm to find an optimal schedule.

3.2.5 Periodic schedules

Periodic schedules are natural schedules that can be expressed in a compact form. As already mentioned, after some possible initialization phase, a periodic schedule repeats the same sequence of checkpoints over and over. Here is the formal definition:

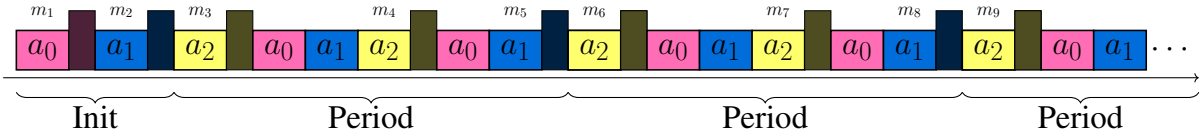


Figure 3.2: A periodic schedule where the period is repeated over time.

Definition 3.4 (Periodic schedules)**.** *A schedule $\mathcal{S} = (m_1, m_2, \dots)$ is periodic if there exists two indices i_0 and k_0 such that for all $i > i_0$, $m_i - m_{i-1} = m_{i+k_0} - m_{i+k_0-1}$.*

An example of periodic schedule is given in Figure 3.2. Intuitively, the schedule enters its steady state after checkpoint number i_0 (with possibly $i_0 = 0$): the period starts right after task number m_{i_0}, and then repeats the same sequence of k_0 checkpoints: the first checkpoint of the period is taken after $m_{i_0+1} - m_{i_0}$ tasks, the second one after $m_{i_0+2} - m_{i_0+1}$ tasks, until the last checkpoint of the period, that of task number $m_{i_0+k_0}$. Then the period repeats indefinitely.

For a periodic schedule, the limit $\lim_{i\to\infty} \mathrm{SD}_i(\mathcal{S})$ always exists, and is given by the slowdown incurrred during each (infinitely repeating) period. Specifically, given i_0 and k_0 in Definition 3.4, we see from Equation 3.1 that this slowdown becomes:

$$\frac{\sum_{i=i_0+1}^{i_0+k_0} \mathbb{E}_\lambda(W_i^{\mathcal{S}}, C_i^{\mathcal{S}}, R_{i-1}^{\mathcal{S}})}{\sum_{j=m_{i_0}+1}^{m_{i_0+k_0}} t_{j-1[n]}}.$$

We prove this result formally below. The major results of this work are the following two theorems, which we prove in Section 3.3 below.

Theorem 3.1. *There exists a periodic schedule that is optimal.*

Theorem 3.2. *We can compute an optimal periodic schedule in polynomial time.*

Proof sketch: The proof has several steps. First we prove that there exists an optimal periodic schedule, i.e., a periodic schedule whose slowdown is minimal. Then we show how to bound the length of the period of this schedule. Once this is done, we have a finite number of periods to look for, and we exhibit a dynamic programming algorithm that determines the optimal period in polynomial time, independently of the number of iterations.

3.3 Optimal checkpoint strategy

In this section, we present several theoretical results and prove Theorems 3.1 and 3.2. Specifically, we start by showing that we can indeed focus on periodic algorithms (Theorem 3.1) in Section 3.3.3. Then in Section 3.3.4, we show that we can compute an optimal periodic schedule in polynomial time (Theorem 3.2).

Beforehand, we introduce the definition of a *pattern* which is at the heart of periodic algorithms (Section 3.3.1), and we present several important properties of patterns in Section 3.3.2.

3.3.1 Paths and patterns

Definition 3.5 (Checkpoint Paths)**.** *A* Checkpoint Path $P = (i_0, [m_1, \ldots, m_{k_P}])$ *is a sequence of* m_{k_P} *tasks* $b_0, \ldots, b_{m_{k_P}-1}$ *such that:*

1. *for* $0 \leq i \leq m_{k_P} - 1$, $b_i = a_{i_0+i[n]}$;
2. *for* $1 \leq j \leq k_P$, b_{m_j-1} *is checkpointed.*

Thus the path starts at task $b_0 = a_{i_0}$ *and includes* m_{k_P} *tasks, up to task* $b_{m_{k_P}-1} = a_{i_0+m_{k_P}-1[n]}$. *The path includes* k_P *checkpoints, including the checkpoint of its last task. The* m_i^{th} *task of the pattern is checkpointed, for* $1 \leq i \leq k_P$. *See Figure 3.3 for an illustration.*

We use the following notations for a path P*: for* $1 \leq i \leq k_P$, *we define* $W_i^P = \sum_{j=m_{i-1}}^{m_i-1} t_{i_0+j[n]}$ *(with the special case* $m_0 = 0$*),* $C_i^P = c_{i_0+m_i-1[n]}$, $R_i^P = r_{i_0+m_i-1[n]}$ *(with the special case:* $R_0^P = r_{i_0-1[n]}$*). We define the length* ℓ *of a checkpoint path* $\ell(P) = \sum_{i=1}^{k_P} W_i^P$ *and its expected execution time, or cost* $\mathcal{C}$ *:*

$$\begin{aligned}\mathcal{C}(P) &= \sum_{i=1}^{k_P} \mathbb{E}_\lambda\left(W_i^P, C_i^P, R_{i-1}^P\right) \\ &= \left(\frac{1}{\lambda} + D\right) \sum_{i=1}^{k_P} e^{\lambda R_{i-1}^P} \left(e^{\lambda(W_i^P + C_i^P)} - 1\right).\end{aligned}$$

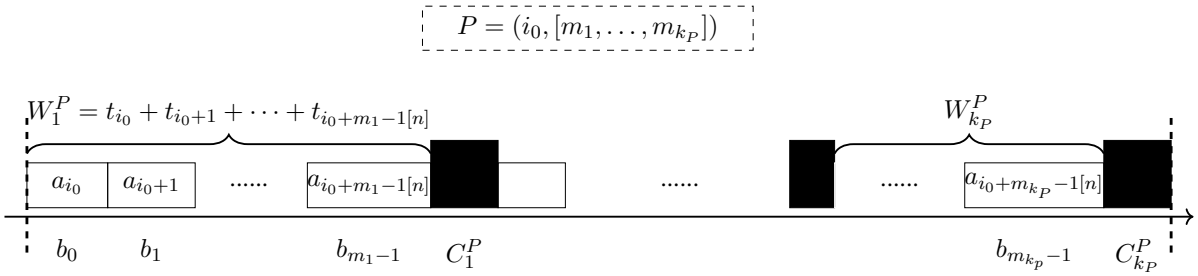


Figure 3.3: Sequence of operations of a checkpoint path $P = (i_0, [m_1, \ldots, m_{k_P}])$. Its length is the sum of its *useful* work (white boxes). Its cost corresponds to its expected execution time if a checkpoint was taken right before its start.

Definition 3.6 (Patterns)**.** *A* Checkpoint Pattern *is a checkpoint path* $P = (i_0, [m_1, \ldots, m_{k_P}])$ *such that* $m_{k_P} = 0[n]$. *Note, for pattern* P*, its length is* $\ell(P) = \frac{m_{k_P}}{n} T$, *where* $T = \sum_{i=0}^{n-1} t_i$.

Such patterns are basic blocks to define periodic schedules. We detail this relation below in Section 3.3.3.

Definition 3.7 (Slowdown of a path (or pattern))**.** *The slowdown of a path is defined as:* $SD(P) = \frac{\mathcal{C}(P)}{\ell(P)}$.

3.3.2 Pattern properties

Using these definitions, we show the following result:

Figure 3.4: Partitioning the schedule into patterns around the checkpoints c_2 (yellow).

Theorem 3.3. *Given a schedule $\mathcal{S}$ of slowdown SD($\mathcal{S}$), there exists a pattern $\mathcal{P}$ such that SD($\mathcal{P}$) $\leq$ SD($\mathcal{S}$).*

Following the proof: Theorem 3.3 aims at showing the existence of a pattern whose slowdown is at most that of any algorithm (hence including optimal algorithms). To do so, we construct a sequence of patterns whose slowdown converges to the requested slowdown (Lemma 3.1) based on sequences taken from the algorithm $\mathcal{S}$. In addition, we impose that this sequence satisfies a *size property*, i.e. that each element of this sequence contains at most n checkpoints (Corollary 3.1). This then helps to find a pattern whose slowdown is exactly that of $\mathcal{S}$.

We start by proving a series of results.

Lemma 3.1. *Given a schedule $\mathcal{S}$, there exists a sequence of patterns $\mathcal{P}_r$ such that, for all $r \in \mathbb{N}$, SD($\mathcal{P}_r$) $\leq$ SD($\mathcal{S}$) $+ 1/r$.*

Proof. Consider a schedule $\mathcal{S} = (m_i)_{i\in\mathbb{N}}$ of finite slowdown SD($\mathcal{S}$). There exists a checkpoint type i_0 which is taken an infinite number of times. We denote by σ_{i_0} the function such that, for all i, $\sigma_{i_0}(i)$ is the i^{th} occurence of checkpoint c_{i_0} in the schedule $\mathcal{S}$ (we set $\sigma_{i_0}(0) = 0$).

In the following, we partition the schedule into paths:

$$\begin{cases} \mathcal{M}_1 = (0, [m_1, m_2, \dots, m_{\sigma_{i_0}(1)}]), \\ \mathcal{M}_i = (i_0 + 1[n], [(m_{\sigma_{i_0}(i-1)+1} - m_{\sigma_{i_0}(i-1)}), (m_{\sigma_{i_0}(i-1)+2} - m_{\sigma_{i_0}(i-1)}), \dots, (m_{\sigma_{i_0}(i)} - m_{\sigma_{i_0}(i-1)})]), \quad \forall i > 1. \end{cases}$$

Intuitively, $\mathcal{M}_1$ is the beginning of the schedule until the first checkpoint of type i_0. Then $\mathcal{M}_2$ is the pattern starting right after and extending up to the second checkpoint of type i_0, and so on. In the definition of $\mathcal{M}_i$, checkpoint indices are shifted to account for the location where the path starts. See Figure 3.4 for an illustration. By construction, each $\mathcal{M}_i$ is indeed a pattern, except for $\mathcal{M}_1$, which is only a path if $i_0 \neq n - 1$.

We now study the slowdown $\text{SD}_{\sigma_{i_0}(i)}$ up to the i^{th} checkpoint of type i_0, i.e., the slowdown of the first i segments. We have

$$\begin{aligned} \text{SD}(\mathcal{S})_{\sigma_{i_0}(i)} &= \frac{\sum_{k=1}^{i} \sum_{j=\sigma_{i_0}(k-1)+1}^{\sigma_{i_0}(k)} T_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})}{\sum_{k=1}^{i} \sum_{j=m_{\sigma_{i_0}(k-1)+1}}^{m_{\sigma_{i_0}(k)}} t_{j-1[n]}} \\ &= \frac{\sum_{k=1}^{i} \mathcal{C}(\mathcal{M}_k)}{\sum_{k=1}^{i} \ell(\mathcal{M}_k)} \\ &= \sum_{k=1}^{i} \alpha_{i,k} \text{SD}(\mathcal{M}_k), \end{aligned} \tag{3.2}$$

where $\alpha_{i,k} = \frac{\ell(\mathcal{M}_k)}{\sum_{j=1}^{i} \ell(\mathcal{M}_j)}$. Hence $\sum_{k=1}^{i} \alpha_{i,k} = 1$, and we have expressed $\text{SD}_{\sigma_{i_0}(i)}$ as a weighted average of the path slowdowns SD($\mathcal{M}_k$).

By definition of $\overline{\lim}$ we have:

- $\overline{\lim_{i\to\infty}}\mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) \leq \overline{\lim_{i\to\infty}}\mathrm{SD}_i(\mathcal{S}) = \mathrm{SD}(\mathcal{S})$;
- For all r, there exists i_r such that $\forall i > i_r$,

$$\begin{aligned}\mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) &\leq \overline{\lim_{i\to\infty}}\mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) + \frac{1}{r}\\ &\leq \mathrm{SD}(\mathcal{S}) + \frac{1}{r}.\end{aligned}$$

Using Equation 3.2, we obtain:

$$\sum_{k=1}^{i} \alpha_{i,k}\mathrm{SD}(\mathcal{M}_k) \leq \mathrm{SD}(\mathcal{S}) + \frac{1}{r}.$$

Since this is a weighted average, it means that there exists k_r, where $1 \leq k_r \leq i$ such that $\mathrm{SD}(\mathcal{M}_{k_r}) \leq \mathrm{SD}(\mathcal{S}) + \frac{1}{r}$. If $k_r \neq 1$, or if $k_r = 1$ and $i_0 = n-1$, we have found the desired pattern by letting $\mathcal{P}_r = \mathcal{M}_{k_r}$. Otherwise, we redo the same proof using the truncated schedule $\widetilde{\mathcal{S}}$ where we delete the first $i_0 + 1$ tasks. Then $\widetilde{\mathcal{S}}$ is a valid schedule for a rotation of the original application, namely for the application $\widetilde{\mathcal{A}} = (a_{i_0+1[n]}, \ldots, a_{i_0})^\infty$, and it has same slowdown as $\mathcal{S}$. The path $\widetilde{\mathcal{M}}_i$ of $\widetilde{\mathcal{S}}$ is the same as the path $\mathcal{M}_{i+1}$ of $\mathcal{S}$ for all i, hence all the paths of $\widetilde{\mathcal{S}}$ are patterns. We then derive the result just as above. □

Lemma 3.2. *For all pattern $\mathcal{P}$, there exists a pattern $\tilde{\mathcal{P}}$ such that:*

1. *SD($\tilde{\mathcal{P}}$) $\leq$ SD($\mathcal{P}$);*
2. *$\tilde{\mathcal{P}}$ contains at most n checkpoints.*

Proof. We show this result by induction on the number of checkpoints in $\mathcal{P}$. Assume $\mathcal{P} = (i_0, [m_1, \ldots, m_k])$, with $k > n$. Then there exists $i_1 < i_2$ such that: $i_0 + m_{i_1}[n] = i_0 + m_{i_2}[n]$ (i.e. the $m_{i_1}^{\text{th}}$ and $m_{i_2}^{\text{th}}$ tasks of the pattern are identical and equal to $a_{i_0+m_{i_1}-1[n]}$).

We now consider the two patterns:

$$\begin{aligned}\mathcal{P}_1 &= (i_0 + m_{i_1}[n], [m_{i_1+1} - m_{i_1}, m_{i_1+2} - m_{i_1}, \ldots, m_{i_2} - m_{i_1}]);\\ \mathcal{P}_2 &= (i_0 + m_{i_2}[n], [m_{i_2+1} - m_{i_2}, m_{i_2+2} - m_{i_2}, \ldots, m_k - m_{i_2},\\ &\qquad m_1 + (m_k - m_{i_2}), m_2 + (m_k - m_{i_2}), \ldots, m_{i_1} + (m_k - m_{i_2})]).\end{aligned}$$

Here, we have decomposed the original pattern into three paths, $\mathcal{P}_{\text{begin}}$ (up to the $m_{i_1}^{\text{th}}$ task), $\mathcal{P}_1$ (from the next task up to the $m_{i_2}^{\text{th}}$ task) and $\mathcal{P}_{\text{end}}$ (from the next task up to the end of the pattern). Now, $\mathcal{P}_2$ is simply the concatenation of $\mathcal{P}_{\text{end}}$ followed by $\mathcal{P}_{\text{begin}}$. See Figure 3.5 for an illustration.

We immediately have:

$$\mathrm{SD}(\mathcal{P}) = \frac{\ell(\mathcal{P}_1)}{\ell(\mathcal{P}_1) + \ell(\mathcal{P}_2)}\mathrm{SD}(\mathcal{P}_1) + \frac{\ell(\mathcal{P}_2)}{\ell(\mathcal{P}_1) + \ell(\mathcal{P}_2)}\mathrm{SD}(\mathcal{P}_2).$$

Because this is a weighted average, then $\min(\mathrm{SD}(\mathcal{P}_1), \mathrm{SD}(\mathcal{P}_2)) \leq \mathrm{SD}(\mathcal{P})$. Each of these patterns has fewer checkpoints than the initial pattern, which concludes the proof. □

Corollary 3.1. *Given a schedule $\mathcal{S}$, there exists a sequence of patterns $\tilde{\mathcal{P}}_r$ for all $r \geq 1$ such that:*

1. *For all r, $\tilde{\mathcal{P}}_r$ contains at most n checkpoints;*

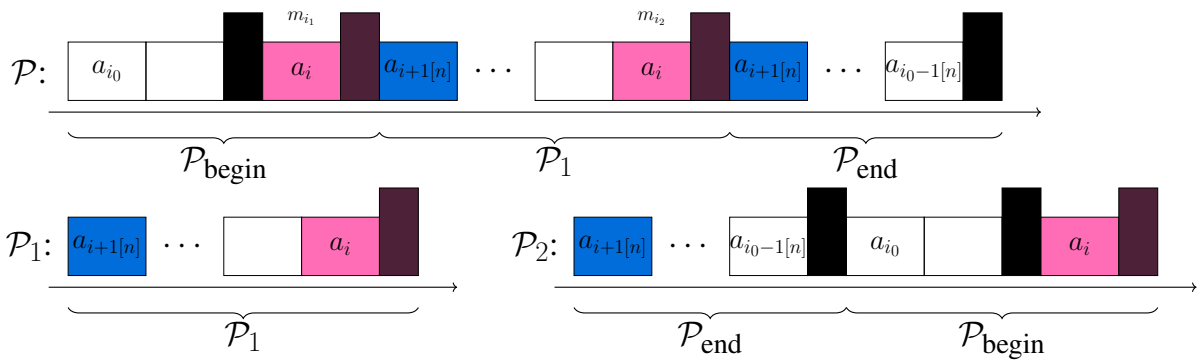


Figure 3.5: From a pattern $\mathcal{P}$ with two identical checkpoints, c_i, to its decomposition into $\mathcal{P}_1$ and $\mathcal{P}_2$.

2. $SD(\tilde{\mathcal{P}}_r) \leq SD(\mathcal{S}) + 1/r$.

This corollary is a direct consequence of Lemma 3.1, for the existence of a sequence that satisfies the slowdown constraint, and of Lemma 3.2, for transforming this sequence into a sequence of patterns that include at most n checkpoints.

Following the proof: At this point, we have constructed a sequence of patterns, whose slowdown converges towards $\text{SD}(\mathcal{S})$. It remains to show the existence of a pattern that reaches the limit. In order to do so, we show that if the number of checkpoints in a pattern is bounded, then the length of the pattern has to be bounded too, otherwise its slowdown would diverge. This is the result shown in Lemma 3.3.

Lemma 3.3. *Given M and k, if $\mathcal{P}$ is a pattern with at most k checkpoints and $SD(\mathcal{P}) \leq M$, then there exists a constant $W_{M,k}$ such that $\ell(\mathcal{P}) \leq W_{M,k}$.*

Proof. Given a pattern $\mathcal{P}$ with k checkpoints, and of length $\ell(\mathcal{P}) = W$, we let $W_1^{\mathcal{P}}, W_2^{\mathcal{P}}, \ldots, W_k^{\mathcal{P}}$ denote the work between its checkpoints. By definition, $\sum_{i=1}^{k} W_i^{\mathcal{P}} = \ell(\mathcal{P})$. Hence there exists i_1 such that $W_{i_1}^{\mathcal{P}} \geq \frac{1}{k}\ell(\mathcal{P})$.

We are now interested in the slowdown of the pattern:

$$\begin{aligned}
\text{SD}(\mathcal{P}) &= \frac{\sum_{i=1}^{k} \mathbb{E}_\lambda(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}})}{\ell(\mathcal{P})} \\
&\geq \frac{\mathbb{E}_\lambda(W_{i_1}^{\mathcal{P}}, C_{i_1}^{\mathcal{P}}, R_{i_1-1}^{\mathcal{P}})}{\ell(\mathcal{P})} \\
&\geq \frac{\mathbb{E}_\lambda(W_{i_1}^{\mathcal{P}}, 0, 0)}{\ell(\mathcal{P})} \\
&= \left(\frac{1}{\lambda} + D\right) \frac{e^{\lambda W_{i_1}^{\mathcal{P}}} - 1}{\ell(\mathcal{P})} \\
&\geq \left(\frac{1}{\lambda} + D\right) \frac{e^{\frac{\lambda}{k}\ell(\mathcal{P})} - 1}{\ell(\mathcal{P})}.
\end{aligned}$$

But $\frac{e^{\frac{\lambda}{k}x}-1}{x}$ tends to infinity when x tends to infinity; hence, because $\text{SD}(\mathcal{P}) \leq M$, we have that $\ell(\mathcal{P})$ is bounded by a function of M and k. Hence the result. □

Proof of Theorem 3.3. We now conclude the proof of Theorem 3.3. From Corollary 3.1, we have a sequence of patterns $\tilde{\mathcal{P}}_r$ with at most n checkpoints and of slowdown $\mathrm{SD}(\tilde{\mathcal{P}}_r) \leq \mathrm{SD}(\mathcal{S}) + 1/r \leq 2\mathrm{SD}(\mathcal{S})$.

From Lemma 3.3, there exists an upper bound such that, for all r, $\ell(\tilde{\mathcal{P}}_r) \leq \tilde{W}$. We show that there is only a bounded number of patterns that satisfy this property:

- Since the length of a pattern is a multiple of $T = \sum_{i=0}^{n-1} t_i$, there are at most $K = \lfloor \frac{\tilde{W}}{T} \rfloor$ possible lengths.
- For a length kT, $1 \leq k \leq K$, there are kn possible checkpoint locations and at most n checkpoints, hence at most $\binom{kn}{n}$ patterns.

Hence in total, the number of possible patterns is upper bounded by $K\binom{Kn}{n}$. The set $\{\mathrm{SD}(\tilde{\mathcal{P}}_r) | r \geq 1\}$ is finite and admits a minimum $S_{\min}$. Let r_0 be one index achieving the minimum: $S_{\min} = \mathrm{SD}(\tilde{\mathcal{P}}_{r_0})$.

Finally, we show that $S_{\min} \leq \mathrm{SD}(\mathcal{S})$: indeed, otherwise there would exist r such that $S_{\min} > \mathrm{SD}(\mathcal{S}) + \frac{1}{r}$ and we would have $\mathrm{SD}(\tilde{\mathcal{P}}_{r_0}) > \mathrm{SD}(\tilde{\mathcal{P}}_r)$, thereby contradicting the minimality. Hence the result. □

3.3.3 Periodic schedules

Using the properties of patterns, we are ready to derive Theorem 3.1. We start by rewriting the definition of periodic schedules using patterns. Indeed, the values i_0 and k_0 from Definition 3.4 allow us to define a pattern that is repeated all throughout the execution. We then select the pattern of minimal length that occurs as early as possible:

Definition 3.8 (Pattern of a periodic schedule)**.** *Given a periodic schedule $\mathcal{S} = (m_1, m_2, \dots)$. Let (k_0, i_0) be the smallest pair (for the lexicographic order) that satisfies: for all $i > i_0$, $m_i - m_{i-1} = m_{i+k_0} - m_{i+k_0-1}$, and $m_{i_0+k_0} - m_{i_0} = 0[n]$. We say that*

$$\mathcal{P}_{\mathcal{S}} = (m_{i_0}[n], [m_{i_0+1} - m_{i_0}, m_{i_0+2} - m_{i_0}, \dots, m_{i_0+k_0} - m_{i_0}])$$

is the pattern of the schedule.

The lexicographic order means that we select first a pattern of minimal length, and in case of a tie, the pattern that starts as early as possible.

Theorem 3.4 (Slowdown of a periodic schedule)**.** *Given a periodic schedule $\mathcal{S}$, its slowdown is equal to the slowdown of its pattern.*

Proof. Given a periodic schedule $\mathcal{S}$, let

$$\mathcal{P}_{\mathcal{S}} = (m_{i_0}[n], [m_{i_0+1} - m_{i_0}, m_{i_0+2} - m_{i_0}, \dots, m_{i_0+k_0} - m_{i_0}])$$

be its pattern. We study the function $\mathrm{SD}_i(\mathcal{S})$ by decomposing the schedule up to its i^{th} checkpoint into three parts: a first part, up to the beginning of the pattern, i.e. up to checkpoint number i_0, then a number $k = \lfloor \frac{i-i_0}{k_0} \rfloor$ of repeating patterns, then a final part whose length is smaller than $\ell(P_{\mathcal{S}})$. The first and final part become negligible as i tends to infinity, hence

$$\mathrm{SD}(\mathcal{S}) = \overline{\lim_{i \to \infty}} \mathrm{SD}_i(\mathcal{S}) = \frac{\mathcal{C}(P_{\mathcal{S}})}{\ell(P_{\mathcal{S}})} = \mathrm{SD}(P_{\mathcal{S}}).$$

□

Proof of Theorem 3.1. Finally, putting everything together, we obtain the final result: Theorem 3.3 states that there exists a pattern P whose slowdown is smaller or equal to that of an optimal schedule. In addition, Theorem 3.4 states that a periodic schedule whose pattern is P has a slowdown equal to that of P, hence it is optimal. □

3.3.4 Finding the optimal pattern

In this section, we show how to compute the pattern of an optimal periodic algorithm. In the following, we say that a pattern $\mathcal{P} = (i_0, [m_1, m_2, \ldots, m_{k_0}])$ is an *optimal pattern*, if it has minimal slowdown.

Bounding the length

> **Following the proof:** In Lemma 3.3, we have shown that it was possible to bound the length of an optimal pattern, which was helpful to prove the existence of an optimal pattern. In order to derive an optimal solution, we want to use a dynamic program whose complexity depends on the number of tasks in a pattern. Unfortunately, the previous bound may lead to a number of tasks in the pattern which is not polynomially bounded. We now show how one can get a tighter bound (Theorem 3.5). At the end of this section, we discuss the size of this bound as a function of the problem instance.

In this section we make intensive use of the following *slowdown function*:

$$f(w, c, r) = \frac{\mathbb{E}_\lambda(w, c, r)}{w} = \left(\frac{1}{\lambda} + D\right) e^{\lambda r} \left(\frac{e^{\lambda(w+c)} - 1}{w}\right).$$

Note that we implicitly used the slowdown function when we defined the slowdown of a schedule. We have the following properties:

Lemma 3.4. *We have the following properties of the slowdown function:*

1. $w \mapsto f(w, c, r)$ *has a unique minimum* w_c *, is decreasing in the interval* $[0, w_c]$ *and is increasing in the interval* $[w_c, \infty)$.
2. $c \mapsto f(w, c, r)$ *(resp.* $r \mapsto f(w, c, r)$*) are increasing functions of* c *(resp.* r*).*

Proof. 2) is obvious. 1) is the result of [16, Theorem 1]. Note that a first-order approximation of w_c is the well-known Young/Daly formula $w_c = \sqrt{\frac{2c}{\lambda}}$ [29, 110]. □

While one might want to use w_c to minimize f, this is only possible for divisible applications. Here, we can checkpoint only at the end of a task, and the amount of work w can only be the sum of some task durations.

Consider a path starting after a checkpoint c_i (hence with a recovery r_i), and ending in a checkpoint c_j. The amount of computation w between these two checkpoints is necessarily of the form

$$W_{i,j}(k) = W_{i,j} + kT, \text{ for some } k \in \mathbb{N},$$

where

1. T is the length of the iterations, $T = \sum_{\ell=0}^{n-1} t_\ell$.

2. $W_{i,j}$ is the length between the end of task a_i and the end of task a_j (possibly of the next iteration), i.e.: $W_{i,j} = \sum_{\ell=i+1}^{j} t_\ell$ (case $j > i$), or $W_{i,j} = T - W_{j,i}$ (case $j < i$), or $W_{i,j} = T$ (case $j = i$).

Additionally, $k \mapsto W_{i,j}(k)$ is an increasing function, hence for all pairs (r_i, c_j), there exists $k^\star_{i,j}$ that minimizes the function $k \mapsto f(W_{i,j}(k), c_j, r_i)$. Let $W^\star_{i,j} = W_{i,j}(k^\star_{i,j})$. Because $f(w, c_j, r_i)$ is decreasing for $w < w_{c_j}$, we have $W^\star_{i,j} - T < w_{c_j}$ (otherwise $W^\star_{i,j} - T$ would be a better solution). Finally, we denote

$$M^\star = \max_i w_{c_i} + T.$$

Then, $M^\star \geq \max_{i,j} W^\star_{i,j}$, and by construction, we have the following property for $M^\star$:

Lemma 3.5. *For all i, j, k_1, k_2 such that $W_{i,j} + k_1 T \geq W_{i,j} + k_2 T \geq M^\star$,*

$$f(W_{i,j} + k_1 T, c_j, r_i) > f(W_{i,j} + k_2 T, c_j, r_i).$$

Finally, we let

$$k^\star = \lfloor M^\star / T \rfloor \tag{3.3}$$

denote the number of iterations that take place during time $M^\star$. We are ready to bound the length between two successive checkpoints within an optimal pattern:

Lemma 3.6. *Given an optimal pattern $\mathcal{P} = (i_0, [m_1, m_2, \ldots, m_{k_0}])$, then for all $1 \leq i \leq k_0$, the inter-checkpoint time between the $(i-1)^{th}$ and the i^{th} checkpoint $W_i^{\mathcal{P}} = \sum_{j=m_{i-1}+1}^{m_i} t_{j-1[n]} \leq 2M^\star$ (using $m_0 = 0$).*

Following the proof: In order to show this result, we show that if the length between two consecutive checkpoints was larger than the bound, then we could add an intermediate checkpoint and create a pattern of smaller slowdown.

Proof. We start by a preliminary property that we use in the following: we show that if the length between two checkpoints is too high, then we can create a pattern of better slowdown by incorporating a checkpoint in the oversized interval.

Given a pattern $\mathcal{P} = (i_0, [m_1, m_2, \ldots, m_{k_0}])$, and given a transformation of this pattern into a pattern $\mathcal{P}'$ of equal length with an extra checkpoint of cost C (and recovery R) located between the $(i-1)^{\text{th}}$ and the i^{th} checkpoint of $\mathcal{P}$, after W units of work, one can verify that

$$\begin{aligned}\text{SD}(\mathcal{P}) - \text{SD}(\mathcal{P}') &= \frac{\mathbb{E}_\lambda\left(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right) - \mathbb{E}_\lambda\left(W, C, R_{i-1}^{\mathcal{P}}\right) - \mathbb{E}_\lambda\left(W_i^{\mathcal{P}} - W, C_i^{\mathcal{P}}, R\right)}{\ell(\mathcal{P})} \\ &= \frac{W_i^{\mathcal{P}}}{\ell(\mathcal{P})} f\left(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right) - \frac{W}{\ell(\mathcal{P})} f\left(W, C, R_{i-1}^{\mathcal{P}}\right) - \frac{W_i^{\mathcal{P}} - W}{\ell(\mathcal{P})} f\left(W_i^{\mathcal{P}} - W, C_i^{\mathcal{P}}, R\right)\end{aligned} \tag{3.4}$$

Indeed, $\ell(\mathcal{P}) = \ell(\mathcal{P}')$, and all inter-checkpoint intervals are identical (and have an equal cost) in $\mathcal{P}$ and $\mathcal{P}'$, except for the interval inside which the extra checkpoint has been added.

We can now prove the result. We show the result by contradiction: assume there exists $i \leq k_0$ such that $W_i^{\mathcal{P}} = \sum_{j=m_{i-1}+1}^{m_i} t_{j-1[n]} > 2M^\star$. We denote by $i_1 = i_0 + m_{i-1} - 1[n]$ and $i_2 = i_0 + m_i - 1[n]$.

• Assume first that $c_{i_2} \geq c_{i_1}$. By monotony, $r_{i_2} \geq r_{i_1}$. We create the pattern $\mathcal{P}'$ such that we add to $\mathcal{P}$ an additional checkpoint after the task of type i_1 at the location $m_{i-1} + k^\star n$ (which indeed

corresponds to a task of type i_1). Then, using the properties of the slowdown function f, and because $W_i^{\mathcal{P}} > k^\star T$, we know that:

$$\begin{aligned} f\left(k^\star T, C_{i-1}^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right) &\leq f\left(k^\star T, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right) && \text{(growth)} \\ &< f\left(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right). && \text{(shape of } f\text{)} \end{aligned}$$

Similarly, $W_i^{\mathcal{P}} > W_i^{\mathcal{P}} - k^\star T \geq M^\star$, then we have:

$$f\left(W_i^{\mathcal{P}} - k^\star T, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right) \leq f\left(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}}\right).$$

Finally, plugging back these values into Equation 3.4, we obtain that $\mathcal{P}'$ has a better slowdown than $\mathcal{P}$, contradicting the optimality.

• Assume now that $c_{i_2} \leq c_{i_1}$. By monotony, $r_{i_2} \leq r_{i_1}$. With a similar demonstration, we show that by including a checkpoint of size c_{i_2} at location $m_i - k^\star n$ (which indeed corresponds to a task of type i_2), leads to the same result. □

Theorem 3.5. *There exists an optimal pattern $\mathcal{P}$ whose length satisfies $\ell(\mathcal{P}) \leq 2nM^\star$, and which includes at most $2n^2(k^\star + 1)$ tasks.*

Proof. From Lemma 3.2, we know that there exists an optimal pattern with at most n checkpoints. Using Lemma 3.6 which gives a bound on the inter-checkpoint time, we obtain the bound on the length. Thanks to Equation 3.3, we know that a length of $M^\star$ corresponds to at most $k^\star + 1$ iterations (of n tasks each), which leads to the bound on the number of tasks. □

We now need to check that $k^\star$ is polynomial in the size of the input. The size of the input is $O(n \max_i \log t_i)$, or equivalently $O(n \log T)$, because the n values t_i are encoded in binary. Here we make the natural assumption that $c_i = O(T)$ and $r_i = O(T)$ for $0 \leq i < n$, meaning that the largest checkpoint/recovery is not longer than a whole iteration[2]. Recall that $w_{c_i} \approx \sqrt{\frac{2c_i}{\lambda}}$ [29, 110], hence $M^\star = O(\sqrt{\frac{\max_i c_i}{\lambda}} + T)$ and $k^\star = O\left(\frac{1}{\sqrt{\lambda T}} + 1\right)$. We obtain a polynomial value $k^\star = O(n \log T)$ as soon as $\mu = \frac{1}{\lambda} = O(T(n \log T)^2)$. This requires that the application MTBF is not too large in front of the iteration length, which makes full sense because otherwise we would not checkpoint more than very rarely, once every many iterations. In Section 3.3.4, we present a dynamic programming algorithm to compute an optimal pattern, whose complexity is polynomial in n and $k^\star$. This complexity is indeed polynomial in the size of the instance under the very natural assumptions that we made.

Computing an optimal pattern

In the previous section, we have shown the existence of an *optimal* pattern of polynomial length. Here, we show how one can compute an optimal pattern through a dynamic programming algorithm. This dynamic programming algorithm relies upon the previous results:

- We study patterns $\mathcal{P}$ of length at most $2nM^\star$ (thanks to Theorem 3.5), and we know that an optimal pattern of this length contains a polynomial number of tasks;
- We consider different initial tasks in the pattern ($i_0 \in \{0, \ldots, n-1\}$);
- We use the fact that there can be at most n checkpoints in the optimal pattern (thanks to Lemma 3.2).

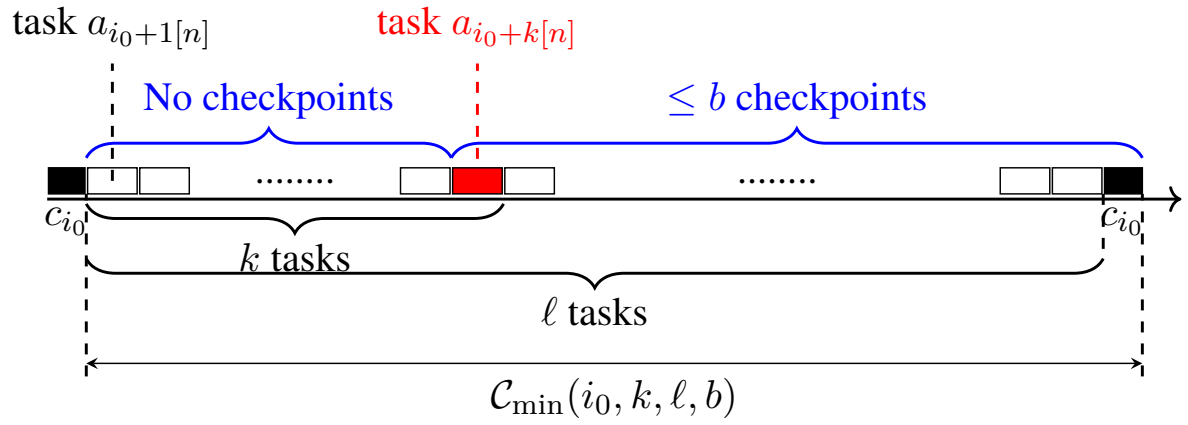


Figure 3.6: Illustration of the minimum expected execution time $\mathcal{C}_{\min}(i_0, k, \ell, b)$ of a series of ℓ tasks as characterized by Lemma 3.7.

The following lemma characterizes the minimal cost of a checkpoint path.

Lemma 3.7 (Minimum cost of a path)**.** *The minimal expected execution time (or cost) of a checkpoint path (i) of ℓ tasks, (ii) whose first task is $a_{i_0+1[n]}$, (iii) with at most b checkpoints, (iv) where the $k-1$ first tasks are not checkpointed and (v) where the last task is checkpointed, is given by;*

$$\mathcal{C}_{\min}(i_0, k, \ell, b) = \min \begin{cases} \mathcal{C}_{\min}(i_0, k+1, \ell, b); \\ \mathbb{E}_\lambda(\sum_{i=1}^{k} t_{i_0+i[n]}, c_{i_0+k[n]}, r_{i_0}) + \mathcal{C}_{\min}(i_0 + k[n], 1, \ell - k, b-1), \end{cases}$$

when $\ell > 0$ and $b > 0$, and where we consider the following initialisation cases:

$$(a) \quad \mathcal{C}_{\min}(i_0, \ell+1, \ell, b) = \begin{cases} 0, & \text{if } \ell = 0, \\ \infty, & \text{otherwise.} \end{cases}$$

$$(b) \quad \mathcal{C}_{\min}(i_0, k, \ell > 0, 0) = \infty.$$

Please refer to Figure 3.6 for a graphical representation of $C(i_0, k, \ell, b)$.

Proof. The result is proven recursively. We start with the initialisation cases. When no checkpoint is allowed ($b = 0$, case (b)), it is not even possible to checkpoint the last task (as required by condition (v), so the cost is infinite. For case (a), $k = \ell + 1$ leads to ℓ tasks not being checkpointed (condition (iv)), which contradicts the fact that the last task is checkpointed (condition (v)), except when the number of tasks is zero: in this case, we assume that no task is performed in this path and no checkpoint is taken.

We now move to the general case. Considering a path that verifies the condition of the lemma, we distinguish two cases:

(i) The k^{th} task is not checkpointed, which leads to the first k tasks not being checkpointed, hence the minimum cost is $\mathcal{C}_{\min}(i_0, k+1, \ell, b)$;

(ii) The k^{th} task is checkpointed. The cost of the first part $k-1$ tasks not checkpointed followed by this k^{th} task and its checkpoint is given by $\mathbb{E}_\lambda(\sum_{i=1}^{k} t_{i_0+i[n]}, c_{i_0+k[n]}, r_{i_0})$. The cost of the rest of the path is recursively expressed as the minimal cost of a path of length $\ell - k$ that starts after task $a_{i_0+k[n]}$ with $b-1$ checkpoints.

[2]Technically, we can relax the assumption to $c_i, r_i = O(T^n)$ without increasing the problem size.

We then select the case that leads to the minimal expected execution time. □

Thanks to Lemma 3.6, we know that in an optimal pattern, there are at most $2n(k^\star+1)$ tasks between two checkpoints. So we can safely restrict our search space to $k = 1, 2, \dots 2n(k^\star+1)$ and consider that the cost for larger values of k is infinite. Hence, the previous recursive definition of the cost is applied to the design of the dynamic programming algorithm (Algorithm 3).

Algorithm 3: Finding the minimum slowdown of a pattern of size at most $2n^2(k^\star+1)$

Procedure PATTERN$(k^\star, n)$**:**
  $maxK \leftarrow 2n(k^\star+1)$
  /* Initialization of ProgDyn */
  for $i_0 = 0$ *to* $n-1$ **do**
    for $\ell = 0$ *to* $2n^2(k^\star+1)$ **do**
      for $b = 1$ *to* n **do**
        if $\ell = 0$ **then**
          $\mathcal{C}_{\min}(i_0, \ell+1, \ell, b) \leftarrow 0$
        else
          $\mathcal{C}_{\min}(i_0, \min(maxK, \ell+1), \ell, b) \leftarrow \infty$
      for $k = 1$ *to* $\min(maxK, \ell)$ **do**
        $\mathcal{C}_{\min}(i_0, k, \ell, 0) \leftarrow \infty$
  /* Precompute $\sum_{i=1}^{k} w_{i_0+i[n]}$ */
  for $i_0 = 0$ *to* $n-1$ **do**
    $W[i_0, 1] \leftarrow t_{i_0+1[n]}$
    for $k = 2$ *to* $2n^2(k^\star+1)$ **do**
      $W[i_0, k] \leftarrow W[i_0, k-1] + t_{i_0+k[n]}$
  /* Computing the ProgDyn */
  for $\ell = 1$ *to* $2n^2(k^\star+1)$ **do**
    for $i_0 = 0$ *to* $n-1$ **do**
      for $k = \min(maxK-1, \ell)$ *downto* 1 **do**
        for $b = 1$ *to* n **do**
          $\mathcal{C}_{\min}(i_0, k, \ell, b) \leftarrow$
            $\min(\mathcal{C}_{\min}(i_0, k+1, \ell, b), \mathbb{E}_\lambda(W[i_0, k], C_{i_0+k[n]}, R_{i_0}) + \mathcal{C}_{\min}(i_0+k[n], 1, \ell-k, b-1))$
  /* Computing the minimal slowdown */
  $SD = \infty$
  $T = \sum_{i=1}^{n} t_i$
  for $i_0 = 0$ *to* $n-1$ **do**
    for $m = 1$ *to* $2n(k^\star+1)$ **do**
      $SD_{\text{temp}} = \mathcal{C}_{\min}(i_o, 1, mn, n)/mT$
      if $SD_{temp} < SD$ **then**
        $SD \leftarrow SD_{\text{temp}}$
  return SD

Theorem 3.6. PATTERN$(k^\star, n)$ *(Algorithm 3) returns the slowdown of the pattern of an optimal periodic schedule with time complexity* $O((k^\star)^2 n^5)$.

Proof. We use the fact that there exists an optimal periodic schedule whose pattern includes a number $m \times n$ of tasks with $m \leq 2n(k^\star+1)$ and uses at most n checkpoints (see Theorem 3.5). Algorithm 3 computes the minimum cost of all patterns including at most this number of tasks, then computes the minimum cost of a pattern whose number of tasks is a multiple of n. The slowdown that we look for is indeed this cost. The complexity of the algorithm derives from the loop nest necessary to recursively compute $\mathcal{C}_{\min}$. □

3.4 Simulation Results

In this section, we describe the experiments conducted to compare the proposed optimal checkpointing strategy with simpler heuristics. We perform simulations on three application scenarios: two from real-life applications (neuroscience and sparse linear solver), and one using synthetic parameters.

The experimental methodology is presented in Section 3.4.1. The results for the neuroscience application are detailed in Section 3.4.2 . The results for the synthetic application are detailed in Section 3.4.3. The results for the sparse linear solver application are detailed in Section 3.4.4. Execution time of the dynamic programming algorithm is presented in Section 3.4.5. Section 3.4.6 provides summary.

3.4.1 Experimental methodology

We detail here the applications, the algorithms used in the simulations and the various settings. All algorithms have been implemented in MATLAB and R. The corresponding code is publicly available at [33].

Neuroscience application

For the first application scenario, we extracted data from a representative neuroscience application, Spatially Localized Atlas Network Tiles (SLANT) [59]. This is an iterative application composed of $N = 10^3$ iterations, and each iteration has $n = 7$ tasks. These tasks are described in Table II, with parameters taken from [43]. Table II reports the mean and standard deviation of the task execution times, which obey a Normal probability distribution. The Pearson correlation of the different tasks was studied in [43], which showed that the tasks are not correlated except for tasks a_0 and a_1 which are proportional. For the first set of experiments in Section 3.4.2, we consider that the tasks are deterministic (as assumed throughout this work) and use mean values as execution times ($t_i = \mu_i$). However, for the second set of experiments in Section 3.4.2, we assess the robustness of our approach and independently draw execution times from the Normal distributions for tasks $a_0, a_2, a_3, a_4, a_5, a_6$, while a_1 is set to be equal to $3.4 \times a_0$ due to its high correlation with a_0. We use a downtime $D = 5$.

Task	a_0	a_1	a_2	a_3	a_4	a_5	a_6
Mean μ_i (sec)	255	871	588	459	3050	804	1130
Stdev σ_i (sec)	96.7	322	76.8	48.1	263	393	568
Checkpoint time c_i (sec)	22.22	61.11	33.33	50	283.33	16.67	61.11
Recovery time r_i (sec)	8.89	24.44	13.33	20	113.33	6.67	24.44

Table II: Tasks of the neuroscience application.

Synthetic application

The second application scenario is randomly generated. We consider an iterative application composed of $N = 10^3$ iterations, each iteration has $n = 10$ or 20 cyclic tasks. We assume that the execution time t_i of each task a_i follows a probability distribution $\mathcal{D}$, where $\mathcal{D}$ is $\textsc{Uniform}(a, b)$. The default instantiation for this distribution is $\mu_{\mathcal{D}} = 550$ for $\textsc{Uniform}(100, 1000)$.

For the first set of experiments in Section 3.4.3, we set checkpoint times as $c_i = \eta t_i$, where η is the proportion of checkpoint time to the execution time of each task. We conduct experiments with $\eta = 0.1$. However, for the second set of experiments in Section 3.4.3, we report results for another instantiation of

checkpoint times, which are then taken in UNIFORM$(10, 100)$, independently of the task running times. In both sets, we use $r = c$ for the recovery time and a fixed downtime $D = 5$,

Sparse linear solver application

The third application scenario is GCR, a Krylov subspace method [37] solving the m-dimensional sparse linear system $A\boldsymbol{x} = \boldsymbol{b}$. Each iteration of the method is divided into n sub-iterations, whose computational and memory requirements increase from one sub-iteration to the next. The common way to control the number of iterative steps within an acceptable range is to adopt a restart strategy [88, 95, 109], that is, to fix a small value n (usually much less than m, such as 10, 20, etc.). If the last n-th sub-iteration does not lead to convergence, then the approximate solution $\boldsymbol{x}^n$ is used as the initial value of a new iteration, and the GCR method is restarted. The process is repeated until a satisfactory approximate solution is found, as detailed in Algorithm 4.

Algorithm 4: GCR(n)

$\boldsymbol{x}^0, \boldsymbol{r}^0 = A\boldsymbol{x}^0 - \boldsymbol{b}, \boldsymbol{p}^0 = P^{-1}\boldsymbol{r}^0, \boldsymbol{q}^0 = A\boldsymbol{p}^0$
for $k = 1, 2, \ldots,$ *until convergence* **do**
  for $i = 0, 1, \ldots, n-1$ **do**
    $\beta = \frac{(\boldsymbol{r}^i, \boldsymbol{q}^i)}{(\boldsymbol{q}^i, \boldsymbol{q}^i)}$ // $4m - 1$
    $\boldsymbol{x}^{i+1} = \boldsymbol{x}^i + \beta \boldsymbol{p}^i$ // $2m$
    $\boldsymbol{r}^{i+1} = \boldsymbol{r}^i + \beta \boldsymbol{q}^i$ // $2m$
    if $\left\|\boldsymbol{r}^{i+1}\right\| \leq \varepsilon$ **then**
      exit
    $\boldsymbol{e} = P^{-1}\boldsymbol{r}^{i+1}$ // $3m - 1$
    $\tilde{\boldsymbol{e}} = A\boldsymbol{e}$ // $2nz(A) - 1$
    for $l = 0, 1, \ldots, i$ **do**
      $\alpha_l = \frac{(\tilde{\boldsymbol{e}}, \boldsymbol{q}^l)}{(\boldsymbol{q}^l, \boldsymbol{q}^l)}$ // $(i+1)(4m-1)$
      $\boldsymbol{p}^{i+1} = \boldsymbol{e} + \sum_{l=0}^{i} \alpha_l \boldsymbol{p}^l$ // $m + (i+1)m$
      $\boldsymbol{q}^{i+1} = \tilde{\boldsymbol{e}} + \sum_{l=0}^{i} \alpha_l \boldsymbol{q}^l$ // $m + (i+1)m$
  $[\boldsymbol{x}^0, \boldsymbol{r}^0, \boldsymbol{p}^0, \boldsymbol{q}^0] \leftarrow [\boldsymbol{x}^n, \boldsymbol{r}^n, \boldsymbol{p}^n, \boldsymbol{q}^n]$

Task	Floating point operations f_i	Vectors to checkpoint	M_i	Vectors to recover			M_i'
a_0	$19m - 4 + 2nz(A)$	$\boldsymbol{p}^1, \boldsymbol{q}^1, \boldsymbol{r}^1, \boldsymbol{x}^1$	$4m$	$\boldsymbol{p}^0, \boldsymbol{p}^1,$	$\boldsymbol{q}^0, \boldsymbol{q}^1,$	$\boldsymbol{r}^1, \boldsymbol{x}^1$	$6m$
a_1	$(6m-1) + 19m - 4 + 2nz(A)$	$\boldsymbol{p}^2, \boldsymbol{q}^2, \boldsymbol{r}^2, \boldsymbol{x}^2$	$4m$	$\boldsymbol{p}^0, \boldsymbol{p}^1, \boldsymbol{p}^2,$	$\boldsymbol{q}^0, \boldsymbol{q}^1, \boldsymbol{q}^2,$	$\boldsymbol{r}^2, \boldsymbol{x}^2$	$8m$
$\ldots$	$\ldots$	$\ldots$	$\ldots$		$\ldots$		$\ldots$
a_{n-2}	$(6m-1)(n-2) + 19m - 4 + 2nz(A)$	$\boldsymbol{p}^{n-1}, \boldsymbol{q}^{n-1}, \boldsymbol{r}^{n-1}, \boldsymbol{x}^{n-1}$	$4m$	$\boldsymbol{p}^0, \ldots, \boldsymbol{p}^{n-1},$	$\boldsymbol{q}^0, \ldots, \boldsymbol{q}^{n-1},$	$\boldsymbol{r}^{n-1}, \boldsymbol{x}^{n-1}$	$(2n+2)m$
a_{n-1}	$(6m-1)(n-1) + 19m - 4 + 2nz(A)$	$\boldsymbol{p}^0, \boldsymbol{q}^0, \boldsymbol{r}^0, \boldsymbol{x}^0$	$4m$	$\boldsymbol{p}^0,$	$\boldsymbol{q}^0,$	$\boldsymbol{r}^0, \boldsymbol{x}^0$	$4m$

Table III: Tasks composing the GCR application.

We consider an iterative application composed of $N = 10^3$ iterations, and each iteration (outer loop k) has either $n = 10$ or $n = 20$ tasks. Each task corresponds to one sub-iteration of the loop on i. The number of non-zero elements of sparse matrix A is denoted as *nz(A)*. We assume that $m = 100000$, $nz(A) = 27m$, and the preconditioner matrix P is a diagonal matrix in the simulation. We pick (somewhat arbitrarily) 27 because it is the size of a $3 \times 3 \times 3$ cube for a neighborhood of interactions, so the matrix has 27 diagonals (3D-stencil for Jacobi or Gauss-Seidel, typically). The number of floating-point operations for task i is $f_i = (6m-1)i + 19m - 4 + 2nz(A)$, see Table III.

We use incremental checkpointing [1, 74] for GCR(n). The vectors that need to be saved if we checkpoint after task i and the corresponding size c_i of the checkpoint are detailed in Table III. Similarly, the vectors that need to be recovered if we experiment a failure task a_i and the corresponding size r_i of the recovery are also detailed in Table III. We observe that c_i remains constant and small for all i, owing to the incremental checkpointing technique. Of course, the larger i, the more vectors to recover, and r_i is increasing.

We consider here that the computing platform has unit speed $s = 1$, so that $t_i = f_i/s = f_i$. In order to test different scenarios for the relative cost of checkpoint compared to computations, we define the Communication-to-Computation Ratio (CCR) as ratio between the cost of communicating one byte to the cost of computing one flop. With the choice $s = 1$, the CCR is exactly the inverse of the bandwidth. Hence, from the size of the memory to checkpoint M_i, we compute the time for a checkpoint: $c_i = M_i \times \text{CCR}$. Similarly, we let $r_i = M'_i \times \text{CCR}$, where the size of the memory to recover is M'_i. We conducted experiments with CCR $\in \{0.1, 0.2, 1, 5, 10\}$, thereby covering a wide range of scenarios (respectively low, balanced and high communication cost).

Failure scenarios

We consider a wide range of failure rates. To allow for consistent comparisons of results across different iterative processes, we fix the probability that a failure occurs during each iteration, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given p_{fail} value, we compute the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda T}$, where T is the execution time per iteration with n tasks. We conduct experiments for five p_{fail} values: $10^{-3}, 10^{-2}, 10^{-1}, 10^{-0.5}$ and $10^{-0.1}$. For each application, these different values of p_{fail} allow us to quantify the risk faced during execution. For example, $p_{\text{fail}} = 10^{-2}$ means one failure will occur every 100 iterations on average. The risk is highest for $p_{\text{fail}} = 10^{-0.1}$ which corresponds to 1 failure per 1.26 iterations on average, while the risk is lowest for $p_{\text{fail}} = 10^{-3}$ which corresponds to 1 failure per $1,000$ iterations on average.

Table IV provides the correspondence between p_{fail} and actual MTBF values for the neuroscience application. The base time (without checkpoint nor failure) for one iteration of the neuroscience application is $7,157$ seconds, or almost 2 hours. Thus $1,000$ iterations will last 83 days approximately. For instance we observe that $p_{\text{fail}} = 10^{-1}$ corresponds to one failure every 19.9 hours, which is typical of several large-scale HPC machines that experience around one failure per day. Smaller values of p_{fail} correspond to platforms with fewer failures, one per week or less. Larger values of p_{fail} represent more failure-prone platforms, with a failure every few hours. Altogether, varying the value of p_{fail} enables to explore a wide range of scenarios.

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
MTBF	82.8 days	8.3 days	19.9 hours	6.3 hours	2.5 hours

Table IV: MTBF for the neuroscience application.

For the synthetic application, task execution times are defined up to a constant factor: we can envision an arbitrary unit of length, ranging from seconds to hours. Then the value of p_{fail} is more representative of the failure rate than the MTBF, whose calculation would need to fix the execution unit. On the contrary, using p_{fail} enables to directly quantify the risk faced by the application in terms of a failure probability per iteration.

For each experiment, the simulations are performed on 100 randomly generated instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_{100}\}$. For all i, an instance $\mathcal{I}_i$ is a pair $(\mathcal{S}_i, \mathcal{F}_i)$, where $\mathcal{S}_i$ (resp. $\mathcal{F}_i$) is the application (resp. failure) scenario associated with the instance. For the neuro-science application, $\mathcal{S}_i$ corresponds to the

values presented in the previous tables, while for the synthetic application scenario, $\mathcal{S}_i$ is randomly generated as described above.

Reference strategies

We consider four reference strategies. The first two strategies are quite natural: (i) CKPTEACHITER consists in checkpointing at the end of each iteration, that is, a checkpoint is taken after the last task a_{n-1} of each iteration; and (ii) CKPTEACHTASK consists in checkpointing after every task a_i of every iteration.

The other two strategies are extensions of the Young/Daly approach for divisible applications where one can checkpoint at any time-step with constant cost c: then the optimal period is to checkpoint every $w_c = \sqrt{\frac{2c}{\lambda}}$ seconds (see Lemma 3.4). For an iterative application, the corresponding approach is to work for w_c seconds and to checkpoint at the end of the current task (and repeat). The difficulty is that c is not well-defined here, because the tasks have different checkpoint costs. With n tasks of checkpoint costs c_i, $0 \leq i < n$, we take the average cost $c_{ave} = \frac{\sum_{0 \leq i < n} c_i}{n}$ and denote the previous strategy using $c = c_{ave}$ as CKPTYDAVE[3]. Finally, the fourth strategy CKPTYDPER is a periodic extension of Young/Daly approach: it chooses the task of an iteration with minimum checkpoint size $c_{\min}$. Only the result of this task will (possibly) be checkpointed. Then it uses the Young-Daly formula to compute how many iterations to include in between two checkpoints, namely $\max\left(1, \text{round}\left(\frac{w_{c_{\min}}}{T}\right)\right)$. We remark that since the checkpoint cost of each task is constant c for the GCR application, the CKPTYDPER heuristic chooses the task with minimum recovery size $r_{\min}$. Only the result of this task will (possibly) be checkpointed. Then it also uses the Young-Daly formula to compute how many iterations to include in between two checkpoints, namely $\max\left(1, \text{round}\left(\frac{w_c}{T}\right)\right)$.

3.4.2 Results for the neuroscience application

We report median values in all experiments, and in the scalability analysis we use boxplots. The color chart is the following: red for CKPTEACHITER, green for CKPTEACHTASK, blue for CKPTYDAVE and purple for CKPTYDPER.

Comparison of the strategies

In Figure 3.7, the makespan of each reference strategy is normalized by the optimal makespan (obtained with Algorithm 3, hence the lower the better. This presentation allows us to directly quantify the performance overhead incurred by each strategy with respect to the optimal approach. Checkpointing after each task, as done by CKPTEACHTASK, gives worst performance when p_{fail} is small (very few failures). Its performance improves significantly when the number of failures increases. This behavior is expected as it is a consequence of the very high number of checkpoints that are taken.

On the contrary, the Young-Daly inspired heuristics (CKPTYDPER and CKPTYDAVE) gives almost optimal results when there are very few failures, and they get worse when the number of failures increases. Again, this behavior is expected: with very few failures, if the frequency of checkpointing is of the same order of magnitude as in the optimal solution, the fact that the checkpointing decision that is taken is not optimal has little impact, because the checkpoint overhead is very low. With numerous failures, CKPTYDPER, which is limited to at most one checkpoint per iteration, does not checkpoint often

[3] We have also experimented with two variants using $c = c_{\min} = \min_{0 \leq i < n} c_i$, and $c = c_{\max} = \max_{0 \leq i < n} c_i$. Results are quite similar.

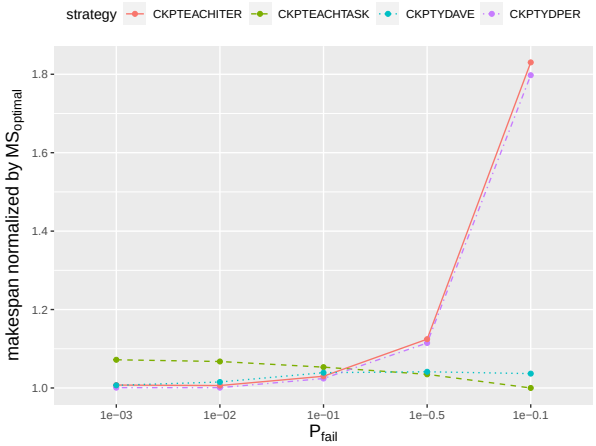


Figure 3.7: Normalized performance overhead with different failure probabilities (neuroscience).

enough, and the loss in work when there is a failure gets too expensive; but CKPTYDAVE can checkpoint more frequently, and its performance degrades less severely, only because it happens to checkpoint some tasks of high checkpoint cost.

Finally CKPTEACHITER is probably the less interesting strategy as its performance is always worse than CKPTYDPER and CKPTYDAVE. As p_{fail} increases, its performance first improves and then gets worse: when p_{fail} is very small, (i) it does not choose the task with smallest checkpoint size and (ii) it checkpoints too often compared to CKPTYDPER and CKPTYDAVE which would allow to checkpoint after several iterations and not just one; conversely, when p_{fail} is very large, checkpointing once after each iteration is not enough, thus the relative cost of the CKPTEACHITER strategy increases. It is still interesting to see that the difference with CKPTYDPER remains always small, while the difference with CKPTYDAVE gets large for frequent failures. It seems that finding the smallest checkpoint size is not critical, while finding the best checkpoint frequency is more important.

For all strategies, when the frequency of failures reaches its maximal value $p_{\text{fail}} = 10^{-0.1}$ (approximately 4 failures every 5 iterations), then all greedy heuristics perform poorly, and the optimal solution provides significant gains, even over CKPTYDAVE which is the best competitor overall.

Absolute overhead

In Figure 3.8, we provide absolute values for the overhead of the strategies of Figure 3.7, for five values of p_{fail}. The time spent for regular periodic checkpointing, or failure-free overhead, is represented in green; it is highest for CKPTEACHTASK, as expected. The failure-induced overhead (downtime, recovery and re-execution) is represented in red; it is higher for CKPTEACHITER and CKPTYDPER. The details of the overheads are interesting: the optimal strategy is really able to trade-off checkpointing and failures; it spends roughly three times less checkpointing than the second best strategy CKPTYDAVE, for a similar failure-induced time. As observed in Figure 3.8, with $p_{\text{fail}} = 10^{-1}$, the cumulated overhead (green and red) ranges from 3.37% to 8.25%, while for $p_{\text{fail}} = 10^{-0.5}$, it ranges from 8.64% to 18.73% .

Scalability

In Figure 3.9, we study the scalability of the approach by varying the number of iterations from 10 to $1,000$. We see that the variance is high at first but the performance of each strategy stabilizes from 100 iterations on.

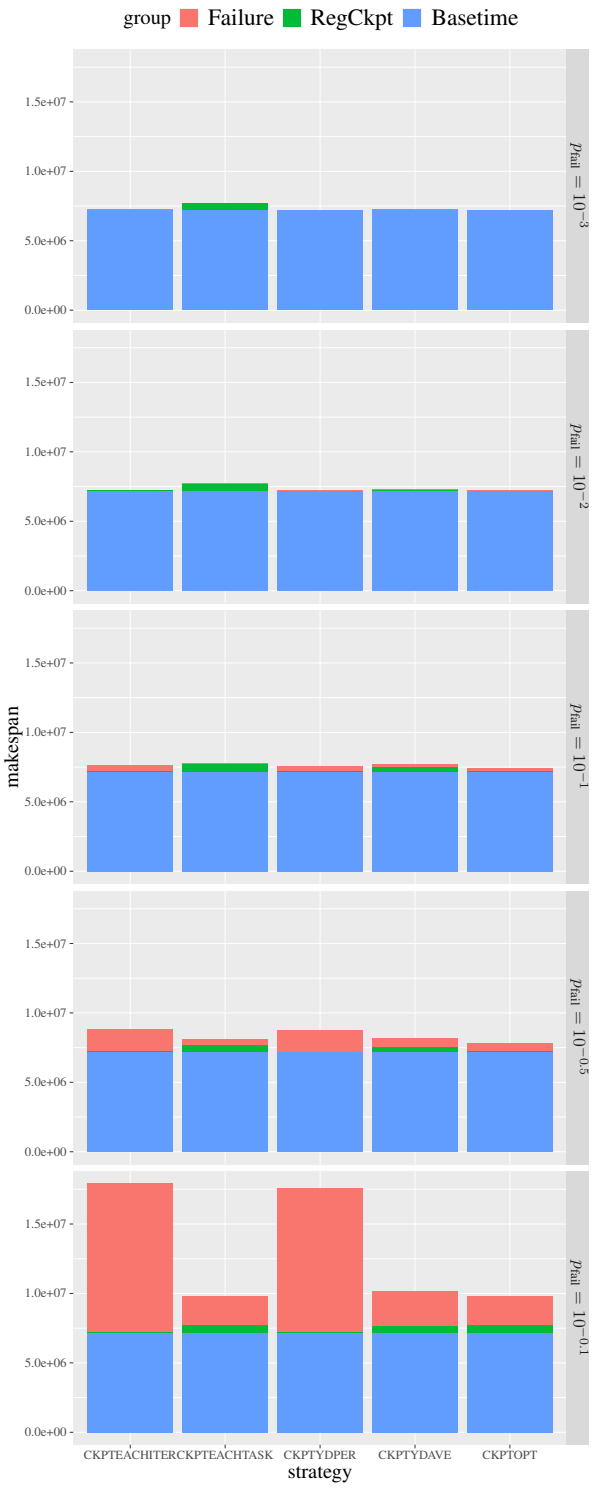


Figure 3.8: Bar plots for absolute overhead (neuroscience).

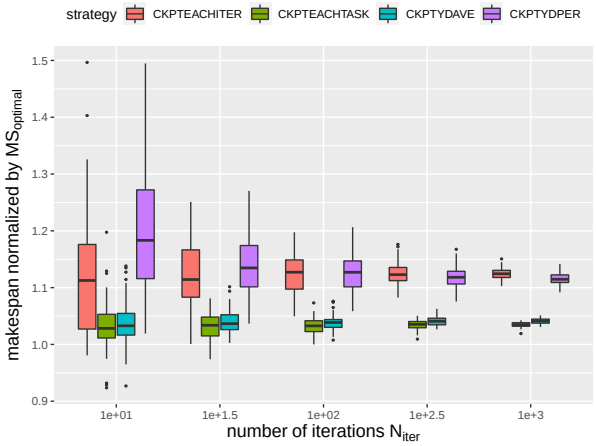


Figure 3.9: Box plots for normalized performance overhead: varying the number of iterations (neuroscience, $p_{\text{fail}} = 10^{-0.5}$).

Robustness

In Figure 3.10, we study the robustness of the approach in front of variations in task execution times, which we draw from their Normal distributions as stated in Section 3.4.1. The results of CKPTEACHITER, CKPTEACHTASK and CKPTYDPER are similar with those in Figure 3.7, with deterministic execution times. However, the results of CKPTYDAVE, which decides on the fly when to checkpoint, become better and close to the optimal strategy when p_{fail} gets smaller.

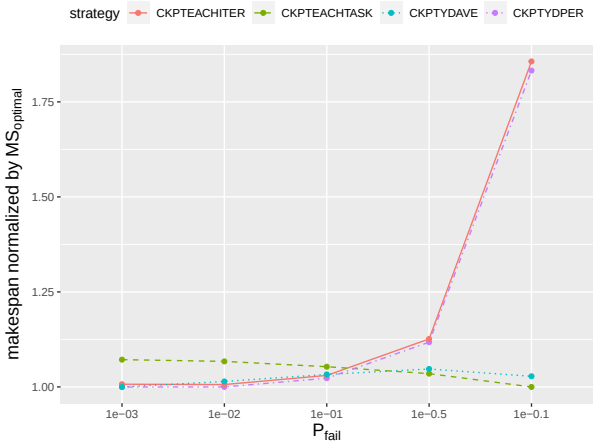


Figure 3.10: Normalized performance overhead with stochastic execution times (neuroscience).

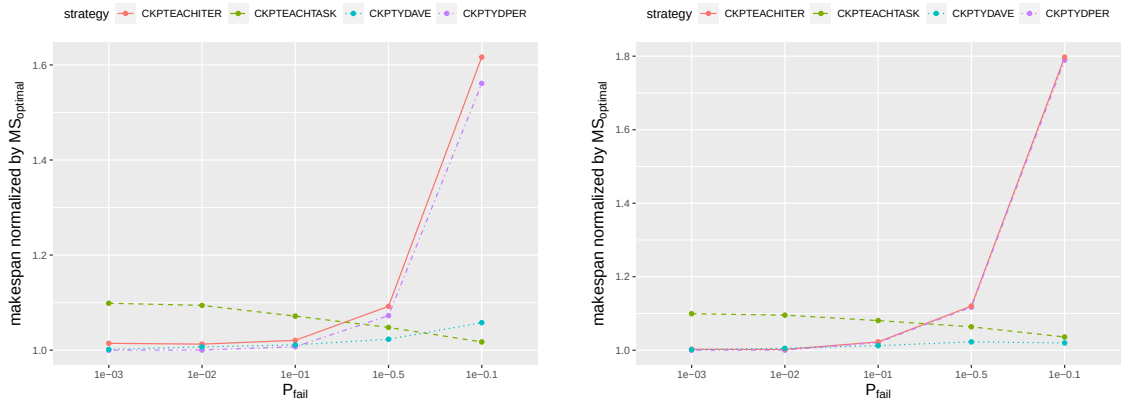
3.4.3 Results for the synthetic application

Comparision of the strategies

Results for the synthetic application scenario are reported in Figures 3.11 and 3.12, with two values of n. For the first set of experiments, we set checkpoint times as $c_i = \eta t_i$, where $\eta = 0.1$ (see Figure 3.11). An important factor that influences the performance of checkpointing strategies, and more precisely of

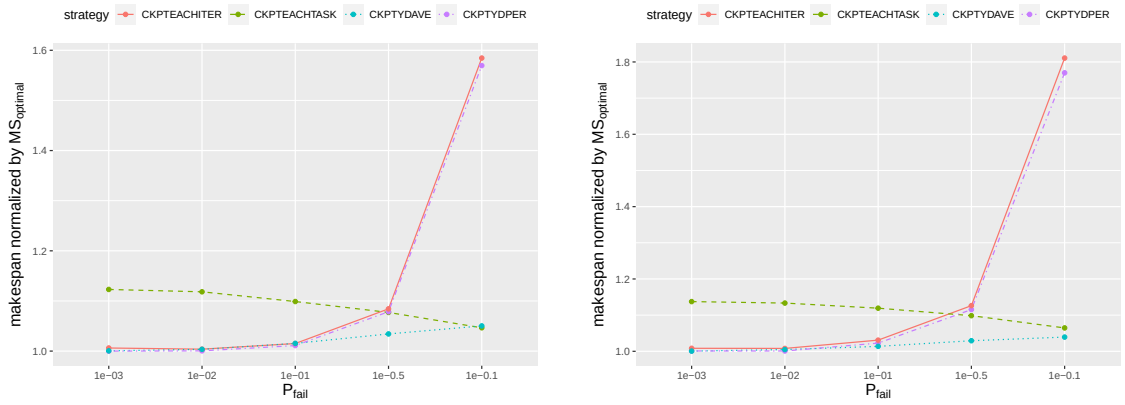
the checkpointing and recovery overheads, is the data-intensiveness of the application. For the synthetic application, in order to test the impact of the correlation between checkpoint costs and task running times on the strategies, we let the checkpoint time move from dependent to independent of the task running time for the second set of experiments (see Figure 3.12).

When n increases from 10 to 20, CKPTEACHITER and CKPTYDPER are closer to the optimal strategy when p_{fail} is small (for 10^{-3} and 10^{-2}), but further away when p_{fail} is large (for 10^{-1}, $10^{-0.5}$ and $10^{-0.1}$); on the contrary, CKPTYDAVE is closer to the optimal strategy for all p_{fail} values. Altogether, the results are quite similar to those obtained with the neuroscience application.



(a) n=10 (b) n=20

Figure 3.11: Normalized performance overhead with different failure probabilities (synthetic, $c_i = 0.1 \cdot t_i$).



(a) n=10 (b) n=20

Figure 3.12: Normalized performance overhead with different failure probabilities (synthetic, c_i drawn in UNIFORM$(10, 100)$).

Absolute overhead

In Figure 3.13, we provide absolute values for the overhead of the strategies of Figures 3.11a and 3.12a, for five values of p_{fail}. The time spent for regular periodic checkpointing, or failure-free overhead, is represented in green; it is highest for CKPTEACHTASK, as expected. The failure-induced overhead (downtime, recovery and re-execution) is represented in red; it is higher for CKPTEACHITER and CKPTYDPER. The optimal strategy is really able to trade-off checkpointing and failures. Altogether, the results are quite similar to those obtained with the neuroscience application.

Impact of the checkpoint time

In the above experiment, we set checkpoint times as $c_i = \eta t_i$ or c_i drawn in UNIFORM$(100\eta, 1000\eta)$, and we conducted experiments with fixed $\eta = 0.1$. Here, we vary the checkpointing cost of each task in order to study its influence on the results. We have two settings for the checkpointing cost. First, we consider that the checkpointing time is proportional to the task execution time, as assumed in the main paper: $c_i = \eta t_i$. We have previously considered $\eta = 0.1$, and we conduct here experiments with $\eta \in \{0.01, 0.05, 0.10, 0.15, 0.20\}$ thereby covering a wide range of scenarios (respectively low, balanced and high checkpointing cost). Lower checkpoint costs can be achieved with state-of-the-art in-memory or hierarchical checkpoint protocols [75], while larger checkpoint costs correspond to traditional protocols that save application data on remote disks. Second, as stated above, we set checkpoint times taken uniformly at random in some interval. We now set checkpoint times taken in UNIFORM$[100\eta, 1000\eta]$ to also cover wider range of scenarios (recall that the average task length is 550 seconds). Results for varying the value of η are reported in Figure 3.14 to Figure 3.18 with the five p_{fail} values and the two checkpoint settings (proportional or uniform).

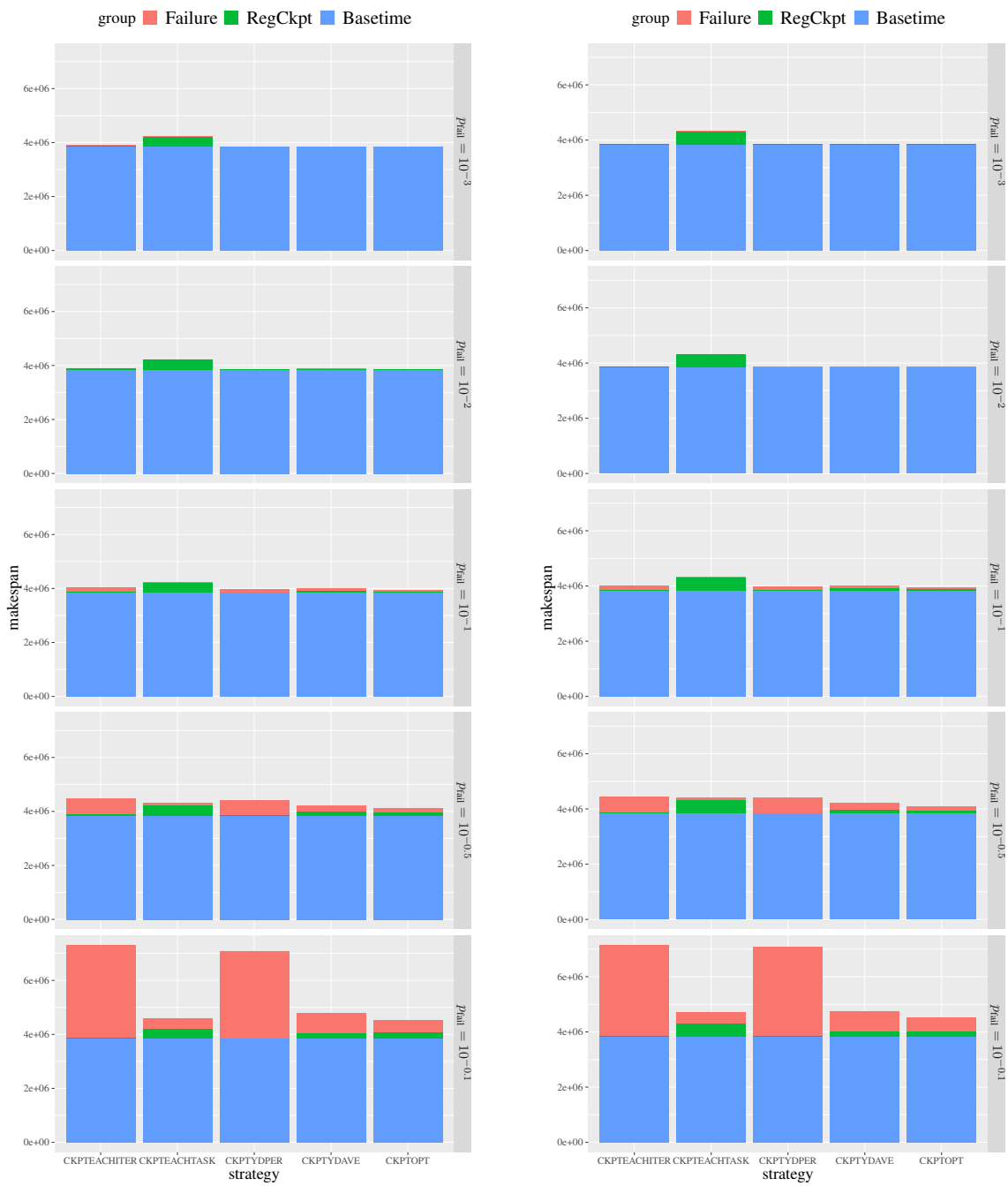
We first observe that the distribution of checkpoint times (proportional to the task execution time or uniform) has very little impact of the results. However, the heuristics do behave very differently when varying checkpoint time, and their behavior also depends on the failure probability. When p_{fail} is small (10^{-3} or 10^{-2}), we note that the CKPTEACHITER heuristic (in red) performs worse and further away from the optimal strategy when η gets larger; for larger values of p_{fail}, its performance is slightly improved when η gets larger. The performance of the CKPTEACHTASK (in green) heuristic becomes worse and further away from the optimal strategy when η gets larger for all p_{fail} values. When p_{fail} is very small (10^{-3}), the performance CKPTYDAVE heuristic (in blue) is quite close with the optimal strategy for all values of η; when p_{fail} is small (10^{-2} or 10^{-1}), it is improved when η gets smaller. Finally, when p_{fail} is large ($10^{-0.5}$ or $10^{-0.1}$), the performance of CKPTYDAVE slightly varies with η, but it is almost optimal for small values of η. The performance of the CKPTYDPER heuristic (in purple) is very close to the optimal strategy when p_{fail} is small (10^{-3} or 10^{-2}). For larger failure probabilities, its performance is improved when η gets larger. Overall, heuristics CKPTEACHTASK and CKPTYDAVE benefits from a very small checkpoint time: when the overhead due to checkpointing is negligible, these heuristics reach an optimal makespan.

3.4.4 Results for the GCR application

Comparison of the strategies

Results for the GCR application are reported in Figures 3.19 to 3.23, with two values of n and five values of CCR.

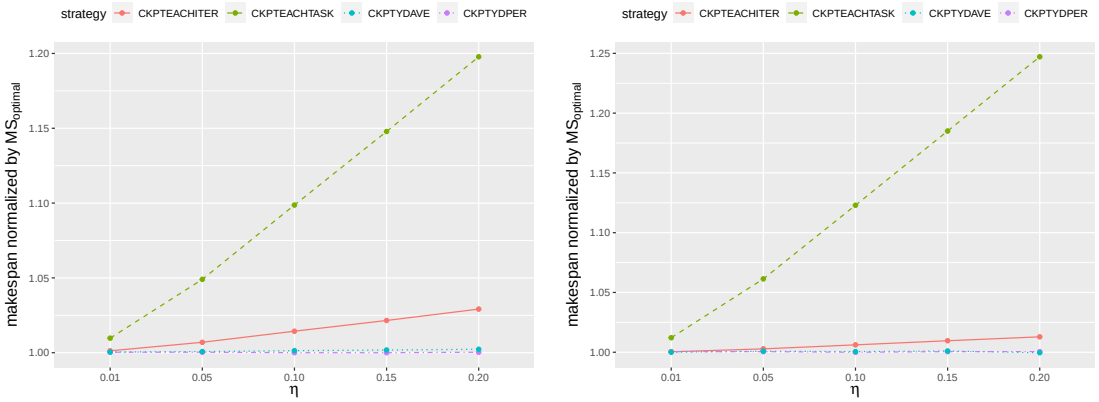
When the CCR increases, CKPTEACHITER and CKPTYDPER are further away from the optimal strategy when p_{fail} is small (for 10^{-3} and 10^{-2}), while both strategies are closer to the optimal strategy



(a) Proportional checkpoint time: $c_i = 0.1 \cdot t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(10, 100)$

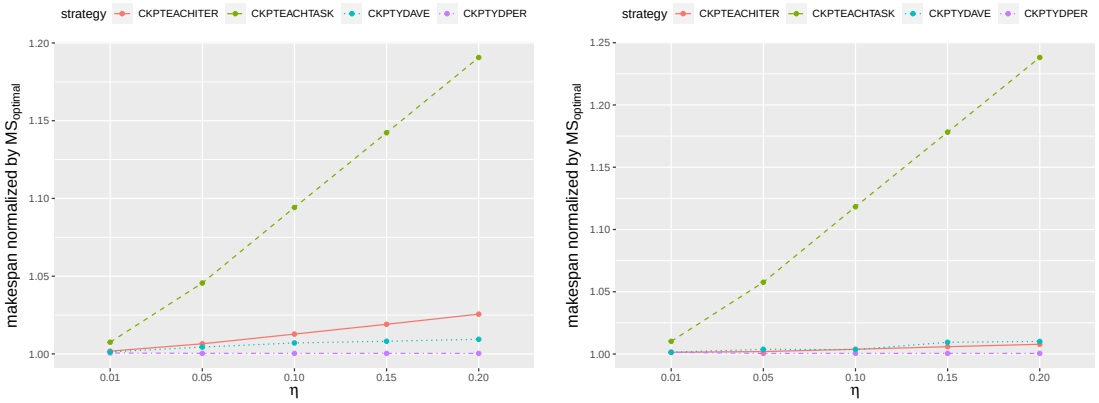
Figure 3.13: Bar plots for absolute overhead (synthetic, $n = 10$).



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

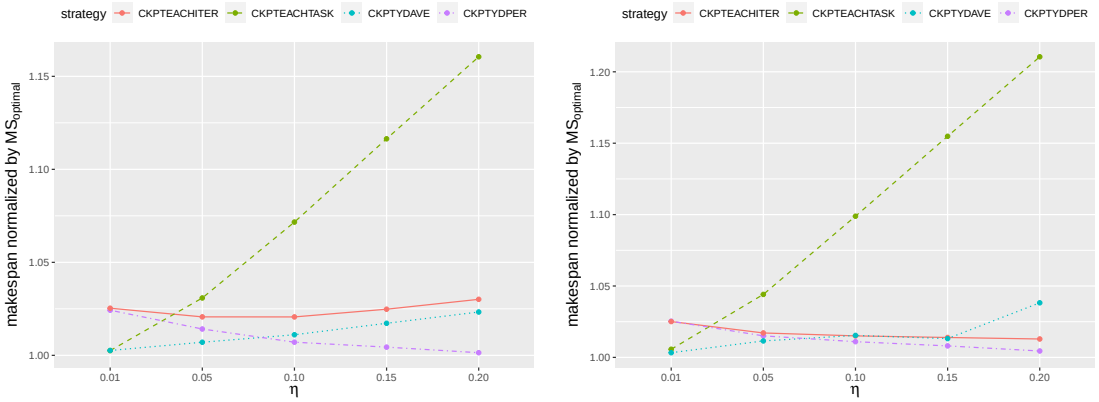
Figure 3.14: Normalized performance overhead with different values of η (synthetic, $n = 10$ and $p_{\text{fail}} = 10^{-3}$).



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

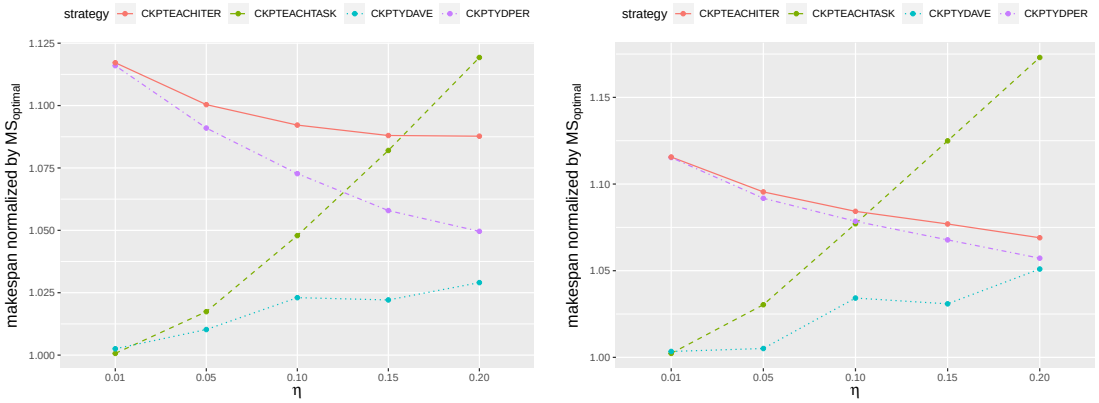
Figure 3.15: Normalized performance overhead with different values of η (synthetic, $n = 10$ and $p_{\text{fail}} = 10^{-2}$).



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

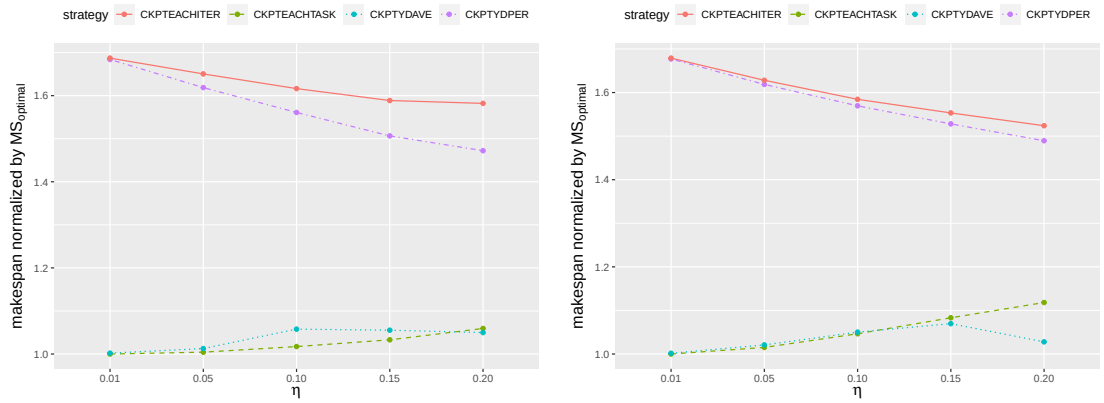
Figure 3.16: Normalized performance overhead with different values of η (synthetic, $n = 10$ and $p_{\text{fail}} = 10^{-1}$).



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

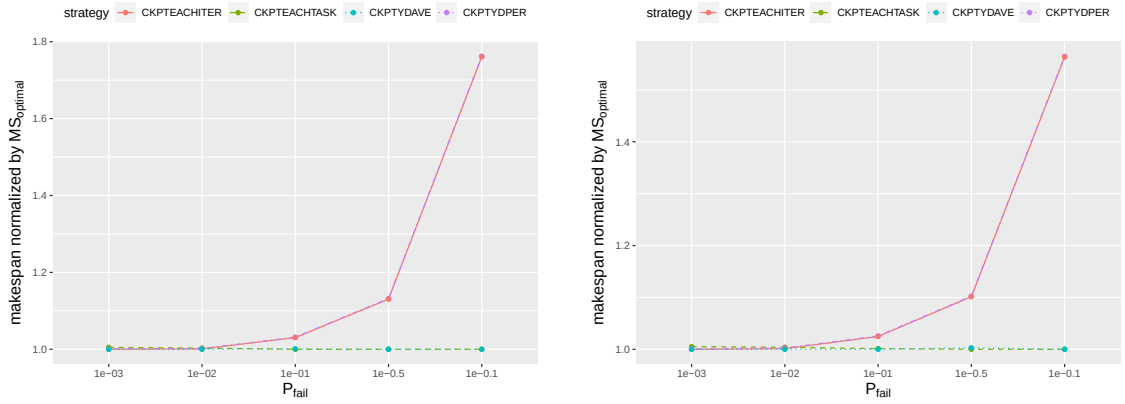
Figure 3.17: Normalized performance overhead with different values of η (synthetic, $n = 10$ and $p_{\text{fail}} = 10^{-0.5}$).



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

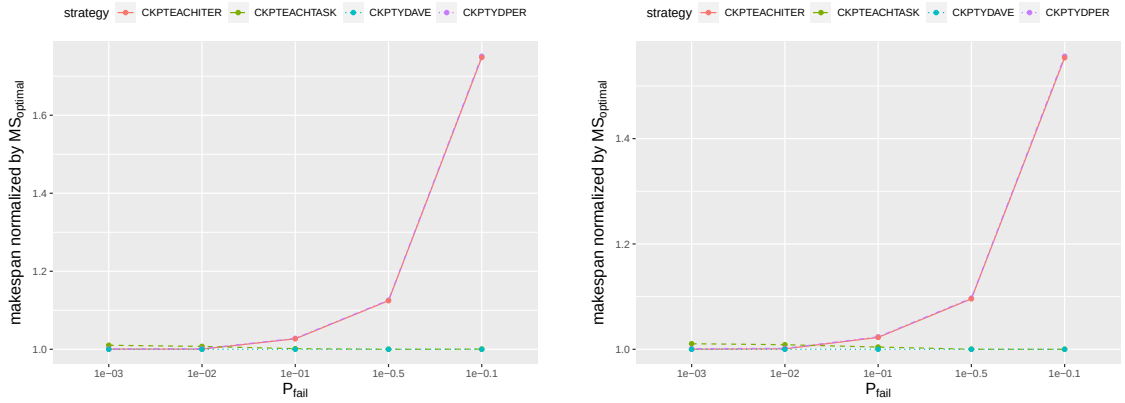
Figure 3.18: Normalized performance overhead with different values of η (synthetic, $n = 10$ and $p_{\text{fail}} = 10^{-0.1}$).



(a) n=10

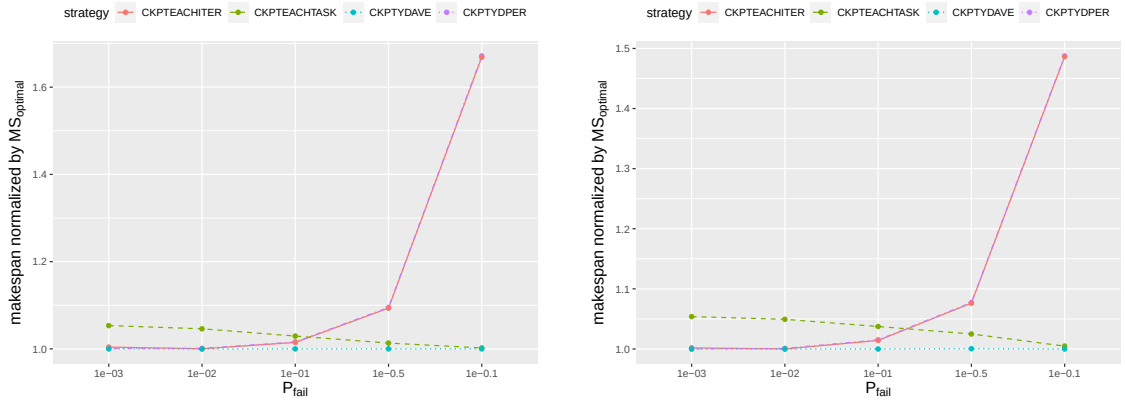
(b) n=20

Figure 3.19: Normalized performance overhead with different failure probabilities. (GCR(n), CCR $=$ 0.1).



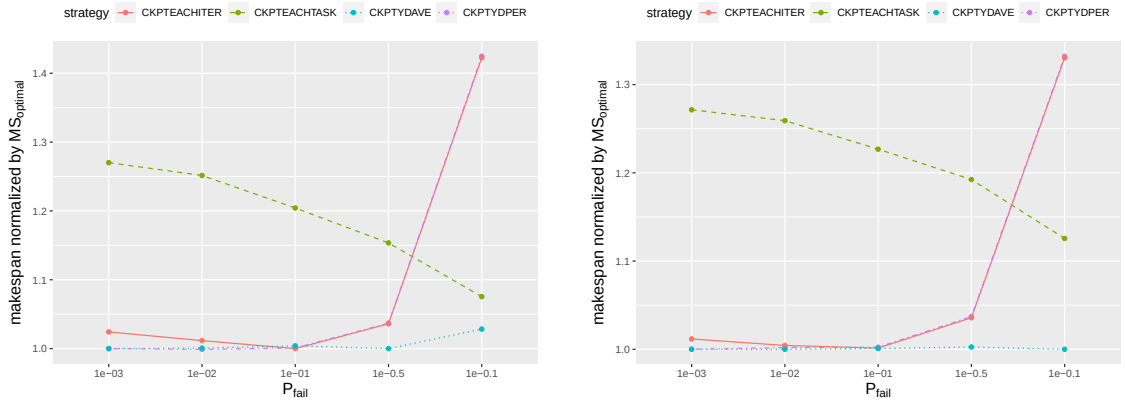
(a) n=10 (b) n=20

Figure 3.20: Normalized performance overhead with different failure probabilities. (GCR(n), CCR $=$ 0.2).



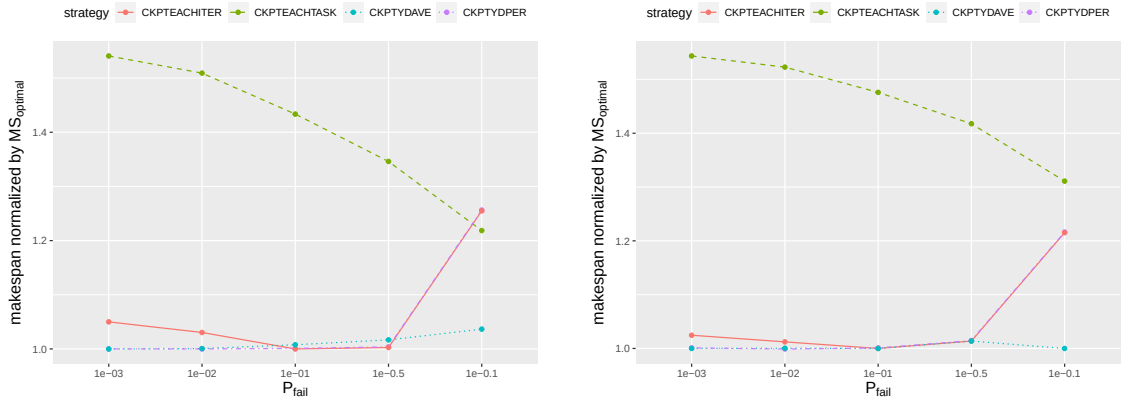
(a) n=10 (b) n=20

Figure 3.21: Normalized performance overhead with different failure probabilities. (GCR(n), CCR $=$ 1).



(a) n=10 (b) n=20

Figure 3.22: Normalized performance overhead with different failure probabilities. (GCR(n), CCR = 5).



(a) n=10 (b) n=20

Figure 3.23: Normalized performance overhead with different failure probabilities. (GCR(n), CCR = 10).

when p_{fail} is large (for 10^{-1}, $10^{-0.5}$ and $10^{-0.1}$); CKPTEACHTASK and CKPTYDAVE are further away from the optimal strategy for all p_{fail} values. In addition, when n increases from 10 to 20, CKPTEACHITER, CKPTYDPER and CKPTYDAVE are closer to the optimal strategy, while CKPTEACHTASK keeps a high overhead with a ratio up to 1.5 to the optimal.

Absolute overhead

In Figure 3.24, we provide absolute values for the overhead of the strategies of Figures 3.19a, 3.21a and 3.23a, for five values of p_{fail}. The time spent for regular periodic checkpointing, or failure-free overhead, is represented in green; it is highest for CKPTEACHTASK, as expected. The failure-induced overhead (downtime, recovery and re-execution) is represented in red; it is higher for CKPTEACHITER and CKPTYDPER. The optimal strategy is really able to trade-off checkpointing and failures. Altogether, the results are quite similar to those obtained with the neuroscience application and synthetic application.

(a) CCR = 0.1 (b) CCR = 1 (c) CCR = 10

Figure 3.24: Bar plots for absolute overhead (GCR(n), $n = 10$).

3.4.5 Execution time of the dynamic programming algorithm

In Table V, we report the execution time of the dynamic programming algorithm for all application scenarios. For the neuroscience application, the execution time is always below one minute. For the synthetic application, the execution time sharply increases when n doubles from 10 to 20, and reaches up to 10 minutes for $p_{\text{fail}} = 10^{-3}$. Table VI, explains why: the number of tasks in the optimal period, estimated by the upper bound of Theorem 3.5, becomes huge, while the actual number of tasks actually occurring in the optimal pattern is much lower. The bound of Theorem 3.5 is overly pessimistic, which increases the execution time of the dynamic programming algorithm. While 10 minutes for the algorithm is negligible in front of the 83 days of the application base time, one could easily decide to use CKPTYDPER, the best reference strategy, instead of the optimal approach. Indeed, for $p_{\text{fail}} \leq 10^{-2}$, their overheads are of the same order.

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
Neuroscience	5.46	0.88	0.21	0.21	0.22
Synthetic, $n = 10$	27.49	5.16	1.27	1.25	1.33
Synthetic, $n = 20$	550.89	95.49	46.75	43.37	46.07

Table V: Execution time (seconds) of the dynamic programming algorithm.

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
Neuroscience: Bound	980	392	196	196	196
Neuroscience: Optimal pattern	14	7	7	7	7
Synthetic, $n = 10$: Bound	1800	800	400	400	400
Synthetic, $n = 10$: Optimal pattern	150	50	50	20	10
Synthetic, $n = 20$: Bound	5600	2400	1600	1600	1600
Synthetic, $n = 20$: Optimal pattern	200	180	20	20	20

Table VI: Number of tasks from the bound of Theorem 3.5 and in the optimal pattern.

3.4.6 Summary

In summary, no reference heuristic is able to give close-to-optimal makespan for every value of p_{fail}: CKPTYDPER is better with very few failures, while CKPTEACHTASK and CKPTYDAVE are better when there are many failures. For these extreme scenarios, using the ad-hoc greedy heuristic is a good solution to trade-off the complexity of finding the solution with the gain in performance. However, in intermediary scenarios, the best reference heuristic can still increase the time to solution by 10% compared to the optimal one, showing the importance of computing the correct solution! Finally, when checkpoint costs can be kept very low, e.g., owing to checkpoint libraries such as VeloC [75], our experiments show that it is safe to use any heuristic that checkpoints sufficiently often, such as CKPTEACHTASK or CKPTYDAVE, because their performance gets close to the optimal solution. Altogether, the best competitors are CKPTYDPER and CKPTYDAVE, but none of them is always superior to the other, while our proposed optimal scheme enables us to carefully optimize the checkpoint pattern for all problem instances.

As for the relevance to exascale HPC scenarios, consider for instance the synthetic application (see Figure 3.13). When p_{fail} is low (10^{-3} to 10^{-1}), all methods are good, except CKPTEACHTASK

whose checkpoint overhead is prohibitive. When p_{fail} increases to $10^{-0.5}$, the overheads of CKPTEACHITER and CKPTYDPER are twice larger than that of CKPTYDAVE, while CKPTEACHTASK achieves intermediate results. Finally, for the highest value $p_{\text{fail}} = 10^{-0.1}$, the best competitor is CKPTEACHTASK, followed by CKPTYDAVE, but the difference with the optimal solution gets much larger for all methods. How realistic is the latter value $p_{\text{fail}} = 10^{-0.1}$ for a future exascale HPC application running on 1 million cores, each with individual MTBF of 10 years? the application will experience a crash every 5 minutes; if an iteration lasts 4 minutes, this corresponds precisely to $p_{\text{fail}} = 10^{-0.1}$.

3.5 Conclusion

In this work, we have investigated checkpointing strategies for iterative applications. Each iteration is composed of a chain of tasks, and these tasks have different lengths and different checkpoint costs. Simple approaches would checkpoint either every task, or the last task at the end of each iteration. An approach inspired by the Young/Daly formula works for P_{YD} seconds, where P_{YD} comes from the Young/Daly formula with a checkpoint cost averaged over all tasks, and then checkpoints as soon as possible (and repeats). Another approach inspired by the Young/Daly formula selects the task with lowest checkpoint cost and checkpoints every p^{th} instance of that task, where p is computed so that the period length approximately obeys the formula. But what is the optimal strategy? The main contributions of this chapter are threefold: (i) we have shown that there exists a periodic strategy that is optimal; (ii) we have provided a dynamic-programming algorithm that computes the optimal period; and (iii) we have shown through a set of experiments that the gains over the other approaches are significant, and that the optimal strategy is the only one achieving a robust solution for all problem instances cases. Given the importance of iterative applications in HPC, we expect that these contributions will greatly improve the deployment of resilient solutions at scale.

This study opens interesting problems in this area, such as dealing with iterative applications whose iterations are composed of a Directed Acyclic Graph (DAG) of tasks, not just a linear chain. Such applications are ubiquitous in real-time systems. However, the mere fact that several tasks may execute concurrently on the platform raises very complicated challenges [50, 51], and most likely only heuristic (suboptimal) algorithms will be obtained.

Chapter 4

Node stealing for failed jobs

4.1 Introduction

Instead of studying resilience for specific applications, such as iterative applications, in this chapter, we study the problem on a larger scale for the whole system and target the batch scheduler. As mentioned in the introduction, batch schedulers have faced additional constraints in the last decade: with reservation sets becoming larger, the frequency of node crashes within the reservation of a job has become more frequent.

When a job fails, the standard policy is to relaunch it as soon as possible (from its last valid checkpoint): the job is put back in the submission queue, but with a high priority, so that it can be re-executed rapidly (e.g., see the 'job failover' section in [91]). If there is a free node available at the time of the failure, the failed job will be able to resume execution (almost) immediately: because it is given a high priority, the failed job will be re-assigned all the surviving nodes of its reservation, plus the free node. Of course it may well be the case that no free node is available at the time of a failure, say if the platform is over-subscribed. In that case, the failed job will have to wait until enough resources become available for its re-execution.

In this chapter, we propose a novel approach for High Performance Computing (HPC) platforms: if there is no free node available when a failure strikes a job, we propose to create one! This means to interrupt another job that is currently executing, and to *steal* one of its nodes and assign it as a new resource to the failed job. This *node stealing* approach is inspired by similar ideas in cloud computing, where users who have paid for *spot instances* [65, 69] can have their resources taken back without prior notification. To the best of our knowledge, this work is the first work that studies node stealing in an HPC framework. There are several decisions to explore:

- **Which job to interrupt?** Clearly, small jobs with one or few nodes are good candidates, because they are easier to re-schedule. But interrupting a small job whose waiting time is already high may not be fair to the owner of that job, so trade-offs between different optimization metrics must be achieved.
- **When to interrupt?** Immediately after the failure is the simplest solution, but the interrupted job will lose the work done since its last checkpoint. Another solution is to wait for a checkpoint before the interruption, or immediately enforce a proactive checkpoint, depending upon what is possible.

The main contributions of this chapter are the following:

- A thorough description of the problem, and how to measure its usefulness;

- A focus on SFSJ (*Steal From Small Jobs*), a strategy which chooses the job to interrupt among those with the smallest number of nodes and, if ties, with the shortest execution time so far;
- An evaluation of SFSJ in a simulated framework, based upon trace-based scenarios;
- A comparative assessment of several other node stealing strategies.

The rest of the chapter is organized as follows. Section 4.2 provides motivational examples. Section 4.3 details the design of the SFSJ strategy. Sections 4.4 and 4.5 are devoted to a comprehensive experimental comparison of SFSJ: Section 4.4 presents the methodology and the various potential objectives, while Section 4.5 presents the results and assesses the efficiency and limits of the approach. Section 4.6 discusses several other node stealing strategies. Finally, Section 4.7 gives concluding remarks and hints for future work.

4.2 Motivation

This section provides a brief motivation for node stealing techniques in the presence of failures. Section 4.2.1 shows that failures dramatically increase the flow of large jobs. Recall that the flow of a job represents the time spent by the job in the system (see Section 4.4.3 for more details on job flows). Section 4.2.2 presents a toy example that explains how the node stealing strategy can be used to decrease these large flows.

4.2.1 Flows and failures

We have simulated an execution of the workload submitted to the Mira platform at Argonne National Laboratory [28, 82, 92] in June 2017 (see details of the simulation in Section 4.4). Figure 4.1 shows different job flows for this execution. The x-axis corresponds to the *job size*: jobs are classified in categories depending on their requested number of nodes. Figure 4.1 (left) shows the maximum flow as a function of a job size, i.e., the maximum flow observed for jobs of a given size. Figure 4.1 (right) shows the mean flow as a function of a job size, i.e., the average flow observed for jobs of a given size. The central value of the boxplot represents the median, while the box extends from the lower to the upper quartiles. The upper whisker extends from the hinge to the largest value no further than 1.5 $\times$ IQR from the hinge, where IQR is the inter-quartile range or distance between the first and third quartiles. The lower whisker extends from the hinge to the smallest value at most 1.5 $\times$ IQR of the hinge. Data beyond the end of the whiskers are called "outlying" points and are plotted individually. Box plots provide flows over five randomly generated failure scenarios.

On both subfigures, a red dot corresponds to the flow obtained in a failure-free environment. We can observe that larger jobs have larger flows in a failure-free environment.

We have enriched the figure with the flows of the same jobs in presence of failures, assuming that the MTBF of the platform is one hour (which is the typical MTBF expected for future scale systems [84]). Given that Mira platform had 49152 nodes, this leads to an individual MTBF for each node of $\mu_{\text{ind}} = 5.61$ years. Failures are randomly generated following a Poisson process (parameter $\lambda = 1/\text{MTBF}$) and several failure scenarios are considered. The results are reported in (black) box plots. Jobs are checkpointed according to the optimal Young/Daly period $P_{YD} = \sqrt{2\frac{\mu_{\text{ind}}}{p}C}$, where p is the job size and $C = 5$ minutes is the (assumed) checkpoint length for all jobs. When a job experiences a failure, it is re-scheduled using the baseline strategy: the job is put back into the queue with highest priority, meaning that it will be re-executed as soon as enough nodes (the job size) are available. If there is a free node available at the time of the failure, this free node can 'replace' the node struck by the failure, and

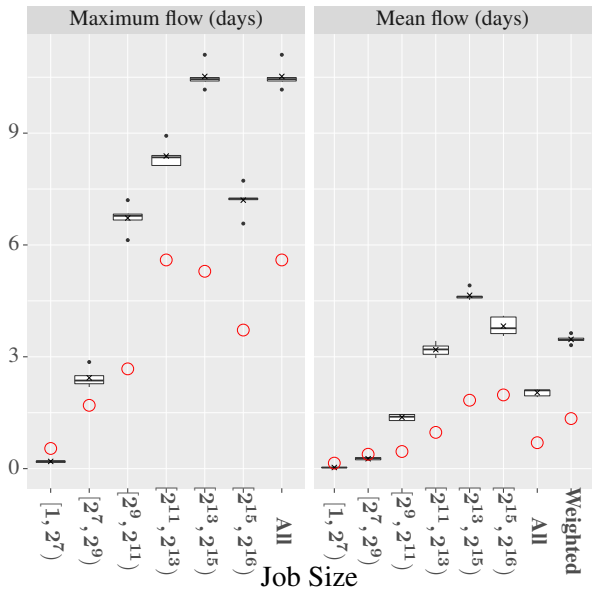


Figure 4.1: Maximum flow and mean flow as a function of job size, without failures (red dots) and with failures (box plots), using BASELINE (workload: Mira, June 2017 [82]). "Weighted" mean flow uses job sizes as weights.

the failed job will be able to resume execution almost immediately. Of course this leads to re-scheduling all the jobs in the execution queue that have not yet started their execution. This baseline strategy is the one used in several batch schedulers such as IBM's LSF [91].

Several observations from Figure 4.1 can be made:

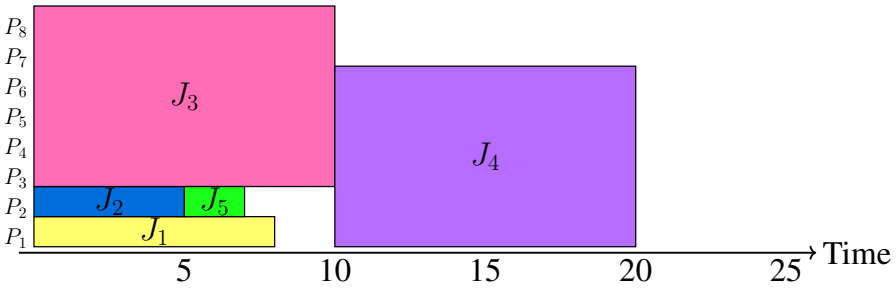
1. the impact of failures is dramatically higher for jobs with more than 2048 nodes, whose flow has increased much more than the flow of jobs with less than 512 nodes. This is because large jobs are harder to re-schedule, due to their high resource demand.

2. the flow of short jobs may be reduced by failures. Indeed, when a large job fails, it has to wait for the completion of another job to get a spare node. During this waiting time, many nodes are left idle and can be used by small and short jobs using backfilling: short jobs are allowed in the "holes" of the schedule; they may start earlier than some (longer) jobs submitted before them, provided that they do not delay these jobs.

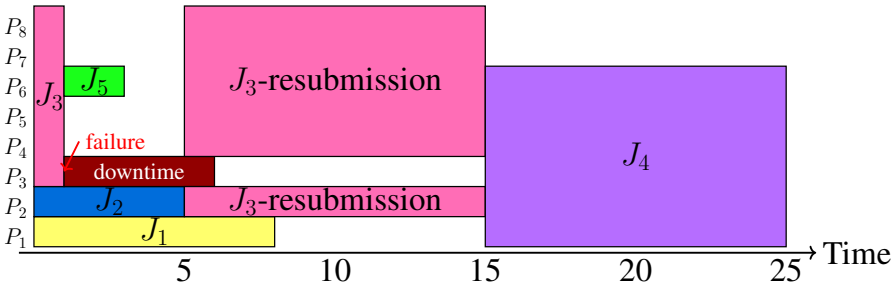
This observation that failures have a non-uniform impact on jobs of different sizes, is at the heart of our approach: would it possible to *steal* nodes from small jobs when large jobs are struck by failures, in order to mitigate the increase of large job flows? A key contribution of this work is to assess the efficiency of *node stealing* in various execution scenarios. Intuitively, if the platform is not over-subscribed, idle nodes will be available most of the time, and node stealing will be rarely (if at all) needed. But as the subscription rate augments, we expect node stealing to become more frequent.

4.2.2 Toy example

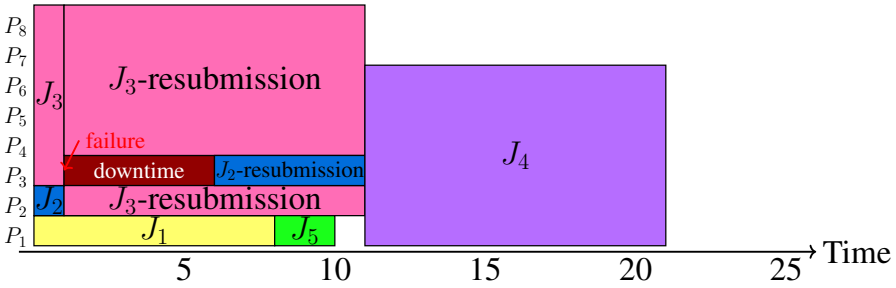
This section uses a toy example to detail the various impacts of node stealing. It provides insight in the decision made throughout this chapter. Consider a platform with 8 nodes. Five jobs are released at time $t = 0$: see Table I and Figure 4.2 for details on these jobs. Since all five jobs are released simultaneously



(a) Fault-free execution



(b) No node stealing



(c) With node stealing

Figure 4.2: Toy example, job details in Table I. Subfigures (b) and (c) assume that a failure occurred at $t = 1$ on P_3.

at time $t = 0$, we can assume that the scheduler has broken ties so that the jobs are scheduled in the order J_1, J_2, ..., up to J_5.

At time $t = 0$, the scheduler starts J_1 on P_1, J_2 on P_2, and J_3 on P_3 to P_8. It reserves P_1 to P_6 for J_4 at $t = 10$. At time $t = 5$, it backfills J_5 on P_2 since it will not delay J_4. Figure 4.2a depicts the fault-free execution.

We consider now that the platform will experience failures. To simplify the example, jobs are not checkpointed and can resume immediately after a failure if there are available nodes, meaning that we neglect any recovery cost. Downtime (rejuvenation time) for each node is $D = 5$, meaning that a node struck by a failure at time t is up again at time $t + 5$. Suppose then that a failure strikes P_3 at $t = 1$. Figure 4.2b depicts the standard scenario. J_3 fails at $t = 1$ and is now the job with the highest priority for re-scheduling. There are only five free nodes at $t = 1$, and this holds true until $t = 5$. Hence J_3 is scheduled for execution at $t = 5$ on nodes P_2 and P_4 to P_8 (since P_3 is unavailable until $t = 6$ due to downtime). Now J_3 completes at time 15 and J_4 completes at time 25. Using backfilling, J_5 is scheduled at $t = 1$ on one available node (P_6 in the figure). In line with the observations made in Section 4.2.1, we see that the smallest job has finished earlier in the presence of a failure than without one, while the large jobs have suffered the most from the failure.

What happens instead if we steal a node when the failure strikes P_3 at $t = 1$? We represent this new scenario in Figure 4.2c. At $t = 1$, we steal P_2 and thereby interrupt job J_2. Job J_3 can re-execute immediately on nodes P_2 (replacing P_3) and P_4 to P_8. J_3 now finishes at time 11. Then J_2 has highest priority and can re-execute on P_3 when is up again at time 6. Now J_2 completes at $t = 11$. Then J_4 is scheduled at time 11 and completes at time 21. Using backfilling, J_5 executes on P_1 when it becomes available.

Table I: Job information for the toy example.

id	Release time	Job size	Job length	Flow without node stealing	Flow with node stealing
J_1	0	1	8	8	8
J_2	0	1	5	5	11
J_3	0	6	10	15	11
J_4	0	6	10	25	21
J_5	0	1	2	3	10

Table I reports some statistics about the flows of the five jobs in the different scenarios without or with node stealing. We see that the flows of the large jobs J_3 and J_4 have decreased, at the price of increasing the flow of the small jobs J_2 and J_5. The maximum value of the flow has decreased from 25 to 21. However its mean value has increased from 11.2 to 12.2. This is interesting as it shows that the mean flow is highly influenced by small jobs, while these jobs are not the most critical jobs on HPC platforms. Another widely used metric is the weighted mean flow, where the mean is weighted by the number of nodes of each job. Here, the weighted mean flow without node stealing is $(1 \times 8 + 1 \times 5 + 6 \times 15 + 6 \times 25 + 1 \times 3)/(1 + 1 + 6 + 6 + 1) = 17$, while the one with node stealing is 14.733.

We also see that the total idle time of the 8 nodes has decreased. Altogether, node stealing seems quite beneficial here! Beyond this toy example, a major contribution of this chapter is to assess the usefulness of node stealing in various realistic execution scenarios.

4.3 Node stealing

This section provides a high-level description of the classic conservative backfilling strategy used by batch schedulers (Section 4.3.1) and details how to extend it to implement node stealing (Section 4.3.2).

4.3.1 Baseline strategy

First-Come First-Serve (FCFS) is a simple approach to submit jobs on parallel supercomputers. However, FCFS often leads to a waste of resources: when there are not enough free nodes for the next job, these free nodes remain waiting until additional nodes become available. A widely-used solution is to use non-FCFS polices, i.e., to allow for a (limited) reordering of the jobs in the queue. *Backfilling* schedulers [73] have been proposed to allow small jobs further away in the queue of waiting jobs to be processed whenever there are enough resources for them. Backfilling may lead to delay some previously allocated jobs, hence it must be controlled so as to guarantee that large jobs will get processed eventually. This is why, in the *conservative backfilling* algorithm, short jobs are moved ahead only if they do not delay any previous job already scheduled.

When a failure hits the system, the remaining part of the job that failed is put back into the scheduling queue, with the highest priority. Depending upon the absence or presence of a resilience mechanism, the remaining part of the job can represent either the whole job or the fraction of the job after the last checkpoint. The schedule is then recomputed with all jobs that have not started yet. If there are multiple jobs that have failed in the queue, they are sorted by non-decreasing arrival time. Throughout this chapter, BASELINE will denote this conservative backfilling scheduling strategy.

4.3.2 Node stealing protocol

Node stealing should be seen as a feature that can be added on top of any batch scheduling strategy. In this work, we add this feature on top of BASELINE scheduling. The core idea is the following: when a failure hits a job (say job J_1), and if there is no (free) node available at the time of a failure, then we select another job (say job J_2) which we *interrupt*. A node from job J_2 is allocated to job J_1, so that job J_1 can resume its execution immediately, either from its last checkpoint if any, or from scratch. Job J_2 is then marked as failed, and it is restarted, again from its last checkpoint if any, otherwise from scratch. The schedule is then recomputed with the following priorities: (high) job J_1; (medium) job J_2; (low) other submitted jobs in the order of the underlying scheduling algorithm (here BASELINE).

In the following sections, we focus on a single node stealing strategy and select the job to interrupt (called *victim* in the following) using the following procedure: *among all running jobs that use the fewest nodes, we select the one that has been submitted the latest*. In other words, the selection criteria are job size first, and job release time to break ties. If no victim job is found with fewer nodes than the failed job, node stealing is not activated. Throughout this chapter, we let SFSJ (*Steal From Small Jobs*) denote this particular node stealing strategy. Other node stealing strategies are discussed and evaluated in Section 4.6, along with the possibility to take a proactive action, i.e., checkpoint the job chosen to be interrupted before actually interrupting it.

We point out that BASELINE and SFSJ behave exactly the same when a free node is available when a job is struck by a failure. Both strategies have the failed job re-submitted with high priority, and therefore start re-execution immediately. However, when no free node is available when a job is struck by a failure, the strategies differ: BASELINE lets the failed job wait until enough resources become available, while SFSJ interrupts another job to be able to restart the failed job immediately.

4.4 Evaluation methodology

In this section, we detail the evaluation setup. Our approach relies on the Batsim simulator [36], which emulates a batch scheduler on a parallel platform (see Section 4.4.1). We have extended Batsim to simulate a failure-prone environment. This extension uses a platform size and a job trace as input. We emulate the Mira supercomputer at Argonne National Laboratory using public traces of this machine [82]. The details of the traces and how they are modified to incorporate resilience mechanisms are presented in Section 4.4.2. Finally, we discuss key objectives used to evaluate the performance of batch schedulers in Section 4.4.3.

4.4.1 Simulation environment

Our simulation environment relies on the existing BatSIM simulator and the Batsched scheduling algorithm toolbox. BatSIM (Batch scheduler SIMulator) [36] is a modular RJMS simulator based on SimGrid [21], which is a state-of-the-art distributed platform simulator. BatSIM is in charge of simulating the behavior of the computational resources. Batsched is a C++ toolbox of scheduling algorithms that take decisions on when and where (which nodes) to schedule a job, and possibly when to interrupt a job. Batsched communicates with BatSIM to receive the information about released jobs and to send scheduling decisions.

There already exists an event injection mechanism in BatSIM/Batsched that allows to make the scheduler aware of external events on the platform. We use and adapt this mechanism in order to simulate node failures and rejuvenation. Whenever Batsched receives the message that a given node has failed, this node is removed from the set of machines available for computations, and thus cannot be used for executing jobs. If a job was running on the node that just failed, Batsched notifies BatSIM that this job is interrupted. Besides, the whole schedule predicted by Batsched has to be recomputed from the current time (the failure time). In the new schedule, the remaining fraction of the job interrupted is given a higher priority, as detailed in Section 4.3.2.

Similarly, when Batsched receives the message that a node that failed before has been rejuvenated, it adds this node to the set of available machines, and the whole schedule is recomputed to take advantage of this newly available resource. Note that in steady-state mode, not all nodes will be up: some have been struck by failures and are rejuvenating. Hence, huge jobs are likely to wait for longer times before execution, and some may fail and be re-submitted several times.

The code developed to run these simulations is publicly available [35], together with Python scripts used to generate failures and R scripts used to handle workloads traces, analyze the results and produce the final plots.

4.4.2 Supercomputer workloads

We use traces of the Mira and Intrepid supercomputers [82] to evaluate the performance of node stealing. Specifically, Batsim uses the following data to compute its schedule: (i) release time; (ii) wall time (predicted execution time of the jobs); (iii) length (actual execution time of the jobs, also called *delay* by Batsim); (iv) number of nodes. We conduct the experiments on two months from the Mira trace: June 2017 and March 2018. These months were selected because their *stress*[1] on the platform are quite reasonable (89.63% and 97.78% respectively), as well as sufficiently different to represent different usage scenarios. Job sizes for both months are detailed in Table II. We also performed experiments on

[1]The stress is defined as the sum of the lengths of jobs submitted this month divided by the total platform availability time in the month, including all nodes.

the June 2013 month from the Intrepid trace. This month is selected as it does not contain any job with less than 2^9 nodes, hence allowing to test our strategy on a quite different workload. Its *stress* on the platform is 89.55%.

Table II: Number of jobs categorized by size (requested number of nodes).

Job size intervals	1	$[2^1, 2^3)$	$[2^3, 2^5)$	$[2^5, 2^7)$	$[2^7, 2^9)$	$[2^9, 2^{11})$	$[2^{11}, 2^{13})$	$[2^{13}, 2^{15})$	2^{15}	49152	total
(Mira) June 2017	8	2	6	10	74	2103	809	269	22	8	3311
(Mira) March 2018	31	3	6	69	117	2481	923	350	31	13	4024
(Intrepid) June 2013	0	0	0	0	0	2001	574	362	31	2	2970

In order to evaluate the impact of failures, we had to transform the traces and control the fault-tolerance mechanism. The full script that takes as input a trace and returns the modified trace is publicly available [35]. The first step is related to the incorporation of failures. Given that Mira platform has 49152 nodes (resp. 40960 for Intrepid), and because we consider failure-intensive scenarios where one or several resources can be down at any time, we reduce the size of the largest jobs from 49152 nodes to 49000 nodes (resp. from 40960 to 40900 for Intrepid). This ensures that no job is rejected because it requires more nodes than actually available on the platform. Finally, we assume that no job is interactive in the traces, for the following two reasons: (i) we cannot distinguish interactive jobs from other jobs in the traces; and (ii) the scheduler would typically exclude interactive jobs from the set of jobs that should be considered in the node-stealing approach.

We can measure the utilization of the platform in a failure-free scenario for this new workload using BASELINE. Unsurprisingly, the utilization is lower than the stress, and notably for March 2018, because of scheduling constraints. We present this data along with statistics about job length in Table III.

Table III: Workload utilization and job lengths.

	Failure-free utilization	Number of jobs	job execution time			
			min	max	mean	median
(Mira) June 2017	88.51%	3311	55s	49.88h	2.64h	0.88h
(Mira) March 2018	92.88%	4024	26s	24.02h	2.79h	1.07h
(Intrepid) June 2013	89.90%	2970	26s	47.08h	2.55h	0.42h

The second step is to add fault-tolerance mechanisms to job submission data. Failures are randomly generated following a Poisson process on each node with parameter λ_{ind}. The Mean Time Between Failure (MTBF) of each individual node is $\mu_{\text{ind}} = \frac{1}{\lambda_{\text{ind}}}$. The MTBF of the whole platform is $\mu = \frac{\mu_{\text{ind}}}{N}$ [55], where N is the total number of nodes of the platform. We assume that the system performs periodic checkpointing using the Young/Daly formula [29, 110]. This means that each job performs a checkpoint every $P_{YD} = \sqrt{2\mu_{\text{job}}C}$ units of time, where C is the time to perform a checkpoint (we use $C = 5$ minutes for all jobs), and μ_{job} is the MTBF for this job. Here μ_{job} is job dependent as it relies on the number of nodes p used by the job: we have $\mu_{\text{job}} = \mu_{\text{ind}}/p$ [55]. Given a periodic checkpoint strategy, the number of checkpoints to be taken linearly depends upon the length of the job. Hence we increase the length of each job accordingly. Furthermore, from a platform perspective, it is only natural to increase the wall time $t_{walltime}$ in a similar way. We compute the new job execution time t_{exec}^{ckpt} and new wall time $t_{walltime}^{ckpt}$:

$$t_{exec}^{ckpt} = t_{exec} + \left\lfloor \frac{t_{exec}}{P_{YD}} \right\rfloor \times C$$

$$t_{walltime}^{ckpt} = t_{walltime} + \left\lfloor \frac{t_{walltime}}{P_{YD}} \right\rfloor \times C$$

During execution, when a failure occurs, jobs are restarted from their last successful checkpoint.

Two key parameters to assess the performance of BASELINE and SFSJ are the downtime and the platform MTBF. We conduct a detailed analysis of the impact of these parameters in Section 4.5.

4.4.3 Measuring performance

When considering the performance of a batch scheduler, there are several metrics to assess. As already stated, from the user's perspective, the most important metric is to minimize the flow, or response time, of the job: *"How fast can I get my results?"*. The flow is defined as the time elapsed from the initial submission of the job up to its completion, possibly after some unsuccessful attempts due to failures. However, from the platform owner's perspective, the most important metric is to maximize the utilization: *"How much work can be executed on the platform per time unit?"*. The utilization is loosely defined as the fraction of time where nodes are doing useful work, i.e., make actual progress in the execution of some jobs. In the following, we provide more details on both metrics and detail how we modified the trace to provide a fair evaluation.

Maximum and mean flow

The flow of a job represents the time spent by the job in the system. The flow is composed of two elements that add up, the waiting time (time elapsed from its submission to the start of its execution) and the execution time (time spent computing with the reserved nodes). If the job fails during execution, it is resubmitted to the scheduler, which usually gives a high priority to re-execution. If the job is checkpointed, only the remaining part of the job after the last checkpoint will be re-executed. Regardless, the job flow accounts for all re-executions and is computed from submission until complete (successful) execution.

Usually, the user makes a reservation with a duration (called *wall time*) and a node count; it is their responsibility to ensure that the reservation has longer duration than the (expected) execution time. This may lead to over-length reservations, in particular when the user is only billed for execution time, not reservation time – a standard scenario on today's platforms. However, longer reservations usually experience a longer waiting time, which is an incentive for users to accurately estimate their reservation length.

Maximum flow is the largest flow for any job running in the system. Mean flow is the average over all jobs in the system. The weighted mean flow is the weighted average over all jobs, where each job is weighted by its size (its number of nodes). This latter quantity gives a higher weight to jobs that use a large number of nodes, which are typically the target jobs deployed onto supercomputers.

Utilization

The utilization is defined as the ratio of the core-hours *occupied to progress a job* over the core-hours *available* during that period. One could expect an utilization close to 1 on a highly-subscribed platform. However, the two main factors that decrease utilization are the following:

1. Idleness due to scheduling: even with sophisticated backfilling techniques, large jobs bring specific constraints to the scheduler; not all nodes can be used at every instant.

2. Failure mitigation: the time spent to checkpoint jobs, to recover from a failure, and to re-execute fractions of jobs that have been lost (after the last checkpoint up to the failure) all decrease platform utilization. In addition, the time spent to re-execute fractions of jobs that have been lost (after the last checkpoint up to the failure) also decreases utilization. It is important to exclude failure

mitigation techniques (such as checkpointing) from the utilization of the platform. Otherwise, an artificial way to increase the utilization would be checkpointing extremely often, hence reducing the waste after each failure.

While idleness due to scheduling has been studied for decades, failure mitigation is a more recent concern. Checkpointing jobs using the Young-Daly formula minimizes the overhead due to failure mitigation. However, resubmitting failed jobs induces an extra burden on the scheduler.

Pruning the traces

Since we simulate a given month of the traces of the Mira platform, the platform is not fully loaded at the beginning of the simulation (first days of the month), and the values for utilization and flow of the jobs that completes are not representative. Similarly, as job submissions stop at the end of the month, the results (utilization and job completion times) are not meaningful after the last submission. Hence we have to carefully select the data used to compute appropriate statistics.

To compute the utilization of the platform as well as the fraction of time spent in various operations (computing, checkpointing, etc.), we define a time window, going from the 11th day of the month up to the 30th of the month when all activities are registered.

When measuring job flow, we cannot use the same time window: by considering only jobs that complete in a predetermined time window, we would not measure the performance of the same subset of jobs for different heuristics. We thus select a slightly different set of data: we order jobs by submission time and remove the first 20% of jobs (intuitively, the ones that are submitted before the platform is fully utilized) as well as the last 20% of jobs (intuitively, the ones that completes later than the last submission time) and compute the flows of all remaining jobs.

4.5 Results

In this section, we describe the experiments that compare node stealing with the baseline strategy (conservative backfilling). As already stated, we perform simulations on the Mira workloads in June 2017 and March 2018, and the Intrepid workloads in June 2013 [82]. In Section 4.5.1, we start by demonstrating the usefulness of the node stealing approach in one specific scenario, for which both the MTBF and the downtime are equal to one hour. This allows us to qualitatively discuss the impact of the strategy. Then we move to a more thorough and quantitative evaluation with varying MTBF and downtime values in Section 4.5.2. In Section 4.5.3, we perform simulations on the synthetic application scenario which is randomly generated.

4.5.1 Baseline Scenario: MTBF=downtime=1 hour

We start this scenario by a remark on the checkpoint of small jobs. With a MTBF of 1h (i.e., $\mu_{\text{ind}} = 5.61$ years), and a checkpoint size of 5 minutes, the Young/Daly period for a job running on 128 nodes is $\sqrt{2 \cdot \frac{5.61 \cdot 365.25 \cdot 24}{128} \cdot \frac{5}{60}} = 8$ hours. This means that any job running on (or on less than) 128 nodes, and which lasts less than 8 hours never checkpoints. In practice, in the timeframe that we are studying, no job of less than 128 nodes performs any checkpoint, and less than 5% of jobs with $[128, 512)$ nodes perform at least a checkpoint. For this scenario, we first discuss platform utilization and then flows.

Table IV: Platform useful utilization for various workloads.

Heuristic	(Mira) 06/2017	(Mira) 03/2018	(Intrepid) 06/2013
BASELINE	79.32%	77.39%	76.71%
SFSJ	80.69%	78.50%	77.26%

Utilization

The utilization is presented in Table IV. With SFSJ, it is 1.4 to 1.7% higher than with baseline scheduling, which is a positive gain, yet limited. To better understand this observation, in Figure 4.3, we report the fraction of total platform time spent into something else than "useful" computations: idle time, resilience mechanisms (checkpoints and restart), downtime, work wasted due to failures (un-checkpointed work when a failure strikes), and any waste due to node stealing (un-checkpointed work interrupted by node stealing and additional recovery time for applications killed).

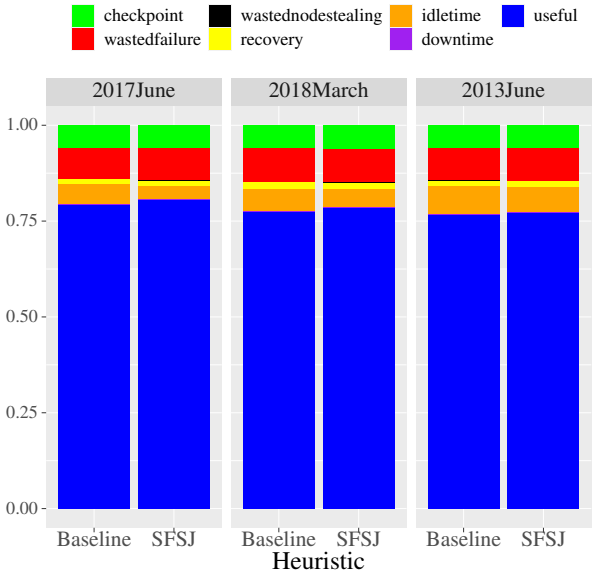


Figure 4.3: Decomposition of platform usage with BASELINE and SFSJ.

Utilization gain can only come from a reduction of the idle time. For BASELINE it corresponds to 5.2% of the platform usage for June 2017 in Mira, (5.8% for March 2018 in Mira and 7.3% for June 13 in Intrepid). Figure 4.3 corroborates the small utilization gains, however it shows that they correspond to relatively important reduction of platform idle time (20% in Mira-March 2018, 40% in Mira-June 2017 and 10% in Intrepid-June 2013). This first item shows that SFSJ is quite impactful given its leeway, especially when the workload contains small jobs.

We further observe that the additional overhead due to SFSJ (work wasted due to job interruption and additional recovery times) is negligible (around 0.1% for all months, as shown by the thin black line on Figure 4.3). This shows that additional resilience mechanisms that one could envision for node stealing (such as proactive checkpoint before interruption) have little room for improvement (see Section 4.6 for a discussion of other node stealing variants).

How often node stealing is actually used? Node stealing is only used when there is no free node available at the time of a failure. Table V provides some key statistics averaged over five randomly

generated failure scenarios. Table Va reports the percentage of time at least one free node is available right after a failure for both approaches. As shown in Table Vb, there is actually a free node available right after a failure, for 84% of failures in June 2017 (Mira), 89% in March 2018 (Mira) and 90% in June 2013 (Intrepid). In this vast majority of cases, node stealing is not activated, and both BASELINE and SFSJ will resubmit the failed job with high priority, hence start its re-execution (almost) immediately. Finally, the different percentages for the reduction of idle time (Figure 4.3) can be explained by the different percentage of situations where SFSJ has to interrupt a job.

Table V: Statistics for the June 2017 and March 2018 Mira workloads.

(a) Percentage of time at least one node is available right after a failure

Heuristic	(Mira) 06/2017	(Mira) 03/2018	(Intrepid) 06/2013
BASELINE	92.09%	93.36%	93.83%
SFSJ	90.67%	92.48%	93.70%

(b) Number of times SFSJ interrupts a job or enrolls a free node available right after a failure

Trace	node stealing	empty node	total failures
(Mira) June 2017	63.4	404.4	467.8
(Mira) March 2018	46.6	416.2	462.8
(Intrepid) June 2013	45.8	410.8	456.6

To conclude from a system performance perspective, there is only little room for improving utilization, and this improvement is duly achieved by SFSJ.

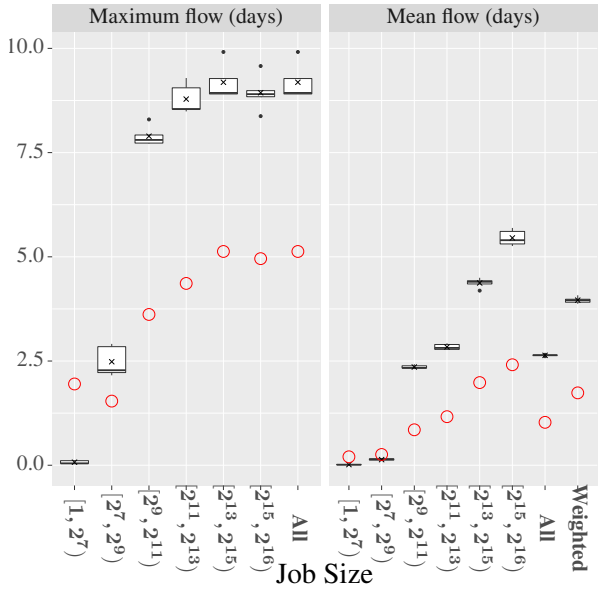


Figure 4.4: Maximum flow and mean flow as a function of job size, without failures and with failures, using BASELINE. Results are for the Mira platform in March 2018.

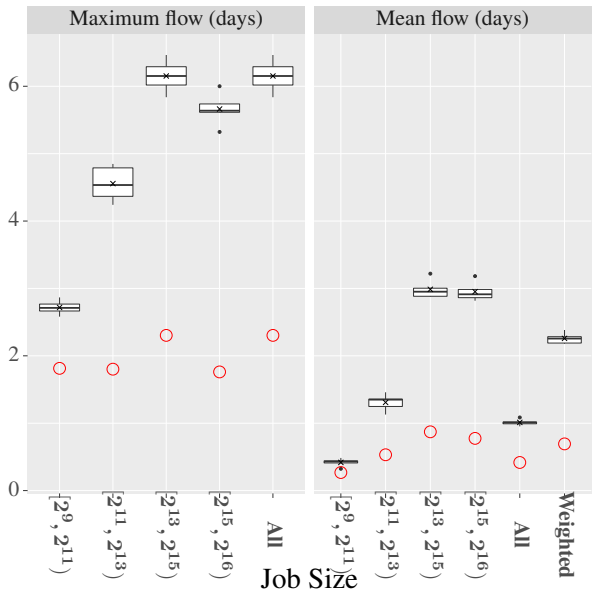


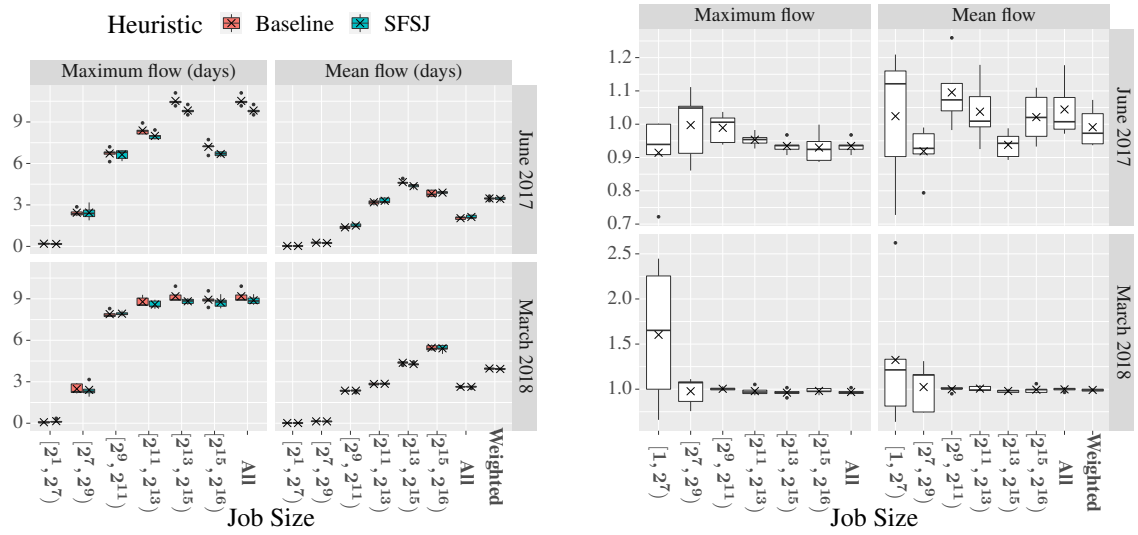
Figure 4.5: Maximum flow and mean flow as a function of job size, without failures and with failures, using BASELINE. Results are for the INTREPID platform in June 2013.

Job flow

How does node stealing impact flows on the platform? First, on a supercomputer, job flows highly depend on the job sizes (i.e., number of requested compute nodes) and on the requested wall times. Indeed, even if the main scheduling algorithm is based on a First-Come-First-Served strategy: (i) backfilling strategies allow to schedule faster "small" jobs that can fit in holes of the schedule; (ii) large jobs are more frequently subject to failures. In Figure 4.4, we plot the response time as a function of the number of requested nodes for BASELINE, without and with failures for March 2018 (Mira). Similar results are shown for June 2017 (Mira) in Figure 4.1 and for June 2013 (Intrepid) in Figure 4.5. In the figure, we report the flow of BASELINE without failure (red dot) and with failures (boxplot). These flows are presented as a function of the node count of the jobs ($x = [2^n, 2^m)$ means that this is the flow of jobs whose number of nodes is in the interval $[2^n, 2^m)$), and also globally ('all", "weighted" on the right of the x axis).

In the failure-free scenario (red dots), we see the impact of backfilling on the flow of jobs: jobs with less than $2^7 = 128$ nodes typically have a much lower flow than larger jobs. The negative impact of failures on the flows is shown by studying the difference between the failure free scenario, and the one with failures: the relative difference is much more important for larger jobs. Interestingly, failures improve the maximum flow of small jobs (jobs with less than 128 nodes). The explanation for this unexpected behavior is that failures create more "gaps" in the schedule to backfill small jobs.

With this in mind, we now compare the various flows between BASELINE and SFSJ. Figures 4.6a and 4.7 show the ratio SFSJ over BASELINE of the flows, hence the lower the better for SFSJ (see Figure 4.6b for the absolute values of these flows for the Mira platform). In these figures, we see that SFSJ significantly improves the maximum flow of large jobs, up to 10% in some scenarios, at the cost of a slight overhead in the flows of small size jobs. In the worst case, the maximum flow of small jobs is increased by a factor 2 (March 2018, Mira), but this needs to be put in perspective: the maximum flow of small jobs is several orders of magnitude lower than the flow of larger jobs. We observe that even when there is no job with a very small number of processors, as with the Intrepid-June 2013 trace, SFSJ



(a) Absolute values

(b) Ratio SFSJ over BASELINE

Figure 4.6: Maximum flow and mean flow as a function of job sizes for Mira. Absolute values are reported in the left plot while ratios (SFSJ over BASELINE) are reported in the right plot. For better visualization of the right plot, an outlier with $x \in [1, 2^7), y = 7.99$ for maximum flow, March 2018, has been hidden.

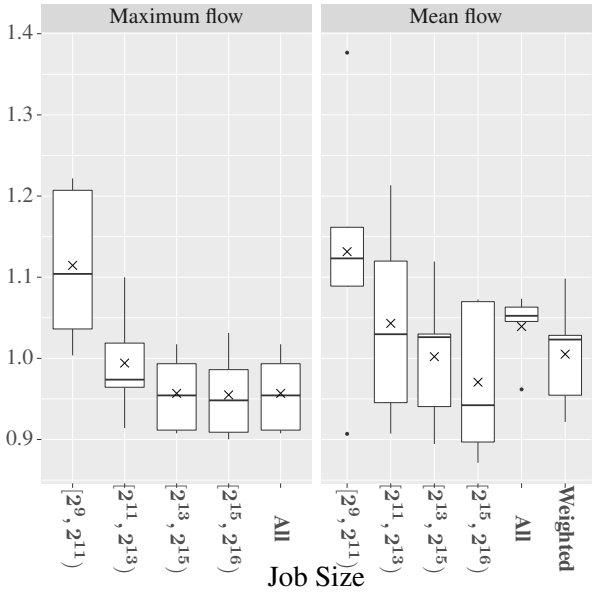


Figure 4.7: Maximum flow and mean flow ratios (SFSJ over BASELINE) as a function of job sizes for Intrepid.

is able to decrease the maximum flow of very large jobs, at the cost of a slight increase (larger than 20%) of the flow of medium size jobs, which is anyway at least twice smaller than that of large jobs. A similar observation is that SFSJ may slightly increase the global mean flow of the platform. Again, this is because this mean flow does not take the size of the jobs into account: if we consider the weighted mean flow instead, where the importance of the job flows depends on the node count, we do observe a decrease when using SFSJ.

Overall, SFSJ significantly improves the maximum flow of large jobs at the detriment of smaller jobs. We argue that this is a good thing since their respective absolute differ by several orders of magnitude.

4.5.2 Quantitative evaluation when MTBF and downtime vary

In the previous section, we have shown the positive impact of SFSJ on a given scenario. We now vary the key parameters, namely the platform MTBF and the duration of the downtime, to fully assess the usefulness of SFSJ and present its limits. We conduct experiments with MTBF $\mu \in \{20\text{min}, 40\text{min}, 1\text{h}, 2\text{h}, 5\text{h}, 10\text{h}\}$, and downtime $D \in \{10\text{min}, 1\text{h}, 1\text{day}\}$.

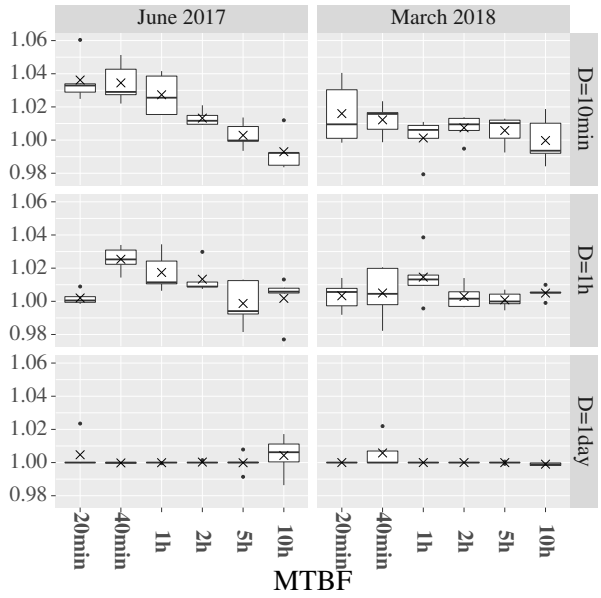


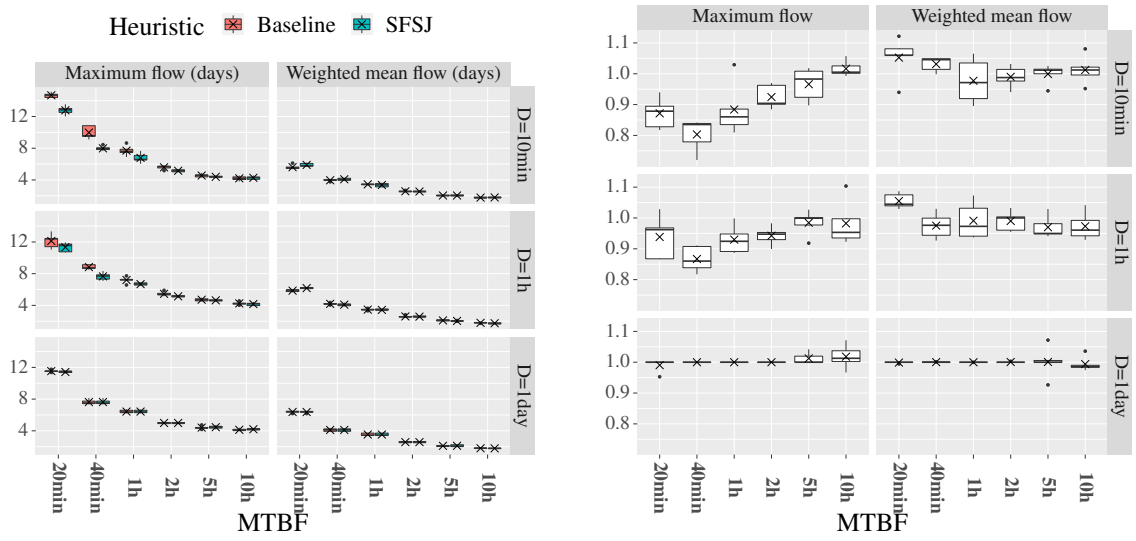
Figure 4.8: Normalized useful utilization as a function of MTBF with failures for Mira. The higher, the better the performance of SFSJ.

Utilization

Figure 4.8 reports the ratio of the utilization of SFSJ over that of BASELINE as a function of the MTBF, and for several downtime values. A value of 1.05 means that SFSJ improves the utilization by 5%, a value of 0.95 that it decreases it by 5%. From a MTBF perspective, the smaller the MTBF (i.e., the more frequent the failures are), the higher the utilization of SFSJ. Similarly, the smaller the downtime, the higher its gain in utilization. With a brief downtime (10min), the improvement of SFSJ is between 2% and 4%, while with a large downtime (1 day), its gain is negligible. This is extremely promising for future supercomputers, whose MTBF decreases linearly with size but whose downtime can (hopefully) be kept at low values.

There is one scenario where we observe a 1% loss in the utilization of SFSJ: June 2017, MTBF of 10h, downtime of 10min. This is a scenario where there are extremely few failures. When there is one, its impact is extremely small (downtime of 10 min) compared to the order of magnitude of the restart time (5 min). Hence in this scenario, killing a small job which does not expect to be killed hurts the system.

To conclude, the more failures, and the smaller the downtime, the more positive impact SFSJ has on platform utilization of the machine. There are some limit scenarios where it may be detrimental (essentially when there are few failures with a small downtime).



(a) Absolute values

(b) Ratio SFSJ over BASELINE

Figure 4.9: Maximum flow for largest jobs and weighted mean flow for all jobs when the MTBF and downtime vary, for June 2017. Absolute values are reported in the left plot while ratios (SFSJ over BASELINE) are reported in the right plot.

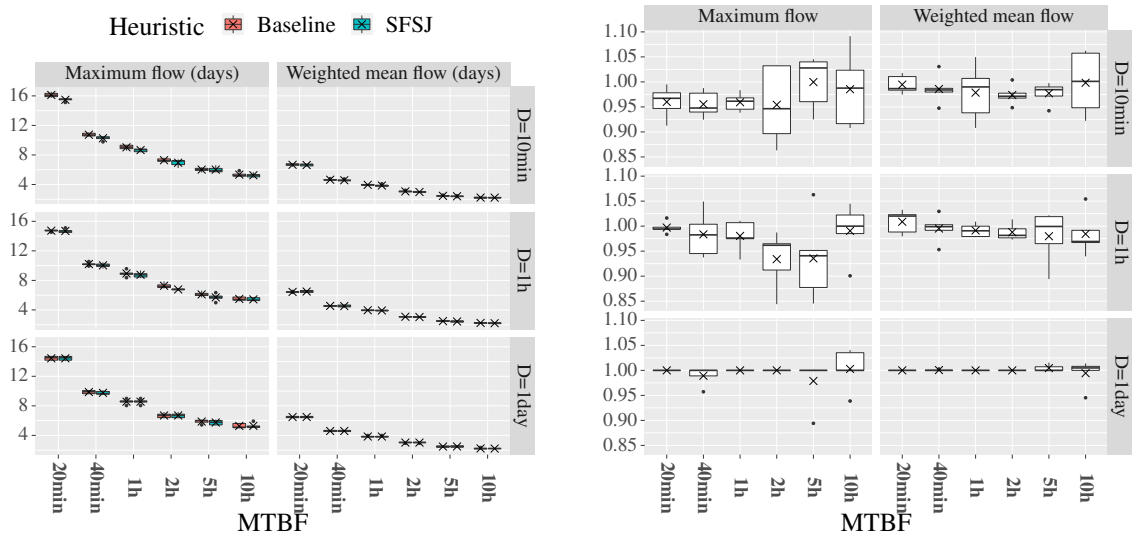
Job flow

In Figure 4.9a, we report the maximum flow of the largest jobs (jobs with more than 2^{15} nodes), and the weighted average flow (overall jobs) as a function of the MTBF for the different scenarios. In Figure 4.9b, we report the ratio of these values (flow of SFSJ over that of BASELINE), hence a value of 1.05 means in this figure indicates that SFSJ increases the flow by 5%, while a value of 0.95 decreases it by 5%. Both figures are for June 2017 (see Figure 4.10 for March 2018).

SFSJ has positive impact on both the maximum flow of the largest jobs and the weighted mean flow over all scenarios. With a small downtime (10 minutes) and a MTBF lower than 5h, the maximum flow improves up to 10-20%. This improvement is not so consequent when the downtime increases, and close to zero when the downtime is equal to 1 day.

4.5.3 Evaluation with synthetic workload

In this section we generate a synthetic workload which allows to create a different job mix. We consider a platform of 128 nodes. We create a set of 1000 jobs with various node requirements between 1 and 64



(a) March 2018

(b) Ratio SFSJ over BASELINE

Figure 4.10: Maximum flow for largest jobs and weighted mean flow for all jobs when the MTBF and downtime vary, for March 2018. Absolute values are reported in the left plot while ratios (SFSJ over BASELINE) are reported in the right plot.

(see Table VI for details). For application A_j, $j \in \{1, \cdots, 1000\}$, we compute its delay t^j_{exec} uniformly at random between 1 minute and 119 minutes (hence an expected time of 60 minutes). Its walltime is $t^j_{walltime} = \alpha_j t^j_{exec}$, where α_j is selected uniformly at random between 1 and 5.

Table VI: There are n_i jobs with $p_i = 2^i$ processors for $i \in \{0, 6\}$ in the synthetic workload.

Number of jobs n_i	504	198	108	65	55	42	28
Job size p_i	1	2	4	8	16	32	64

Table VII: Utilization for synthetic trace.

Heuristic	Synthetic
BASELINE	70%
SFSJ	72%

Jobs arrive in the system in a random order, with an inter-arrival time following an exponential distribution of mean λ (Poisson process). The value of λ is set to $174s$ so that the stress of the system is equal to 95%, i.e.:

$$\frac{1}{\lambda} \cdot \frac{\sum_i n_i \cdot 2^i \cdot 3600}{128 \cdot 1000} = 0.95.$$

Finally, we consider a platform MTBF of 30 minutes, a checkpoint and recovery time of 5 minutes and a downtime of 10 minutes.

Results are presented in Table VII and Figure 4.12. They are similar to those observed in the previous sections: we observe an improvement of 10-15% on the maxflow and meanflow of large jobs. As a trade-off, the maximum flow of the smallest jobs increases by a large factor. However, the flow of small jobs in absolute value is much lower than that of the largest jobs (see Figure 4.11).

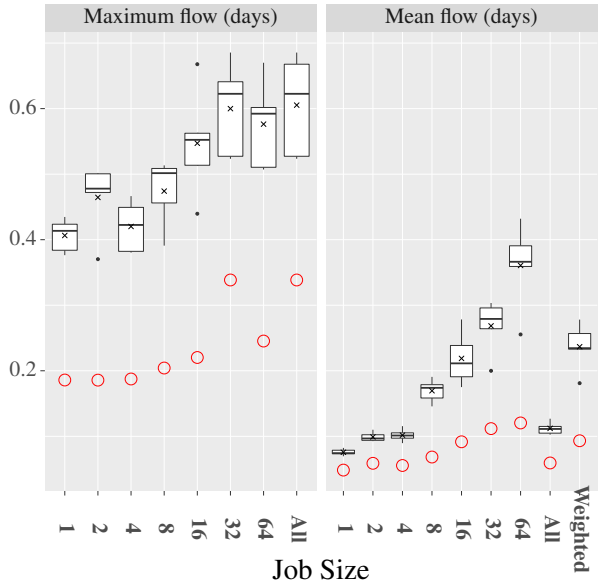


Figure 4.11: Maximum flow and mean flow for BASELINE for the synthetic workload as a function of job size, without failures and with failures.

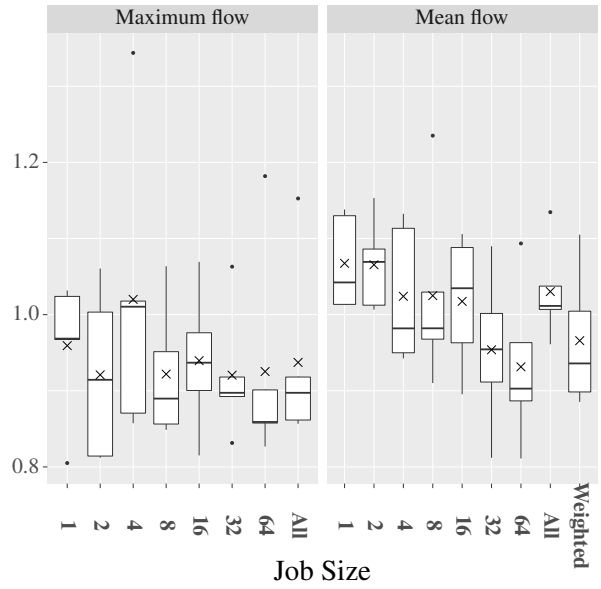


Figure 4.12: Maximum flow and mean flow ratios (SFSJ over BASELINE) as a function of job sizes for synthetic workload.

To conclude this experimental evaluation, SFSJ has positive impact on both the maximum flow of large jobs and the platform utilization of the machine, as soon as failures are not too infrequent (the very framework for which SFSJ is introduced). The impact is greater when the downtime is small.

4.6 Additional heuristics

In the previous section, we have focused on SFSJ, the node stealing heuristic which interrupts the job with the fewest nodes, and which has been running for the smallest amount of time if there is a tie. In this section, we have designed and implemented many other variants, and compared their performance with SFSJ. In a nutshell, here are several design choices that we considered:

1. *victim* job: node stealing can interrupt either the (currently running) job with the smallest number of nodes, or the one with the latest release time;

2. *when to interrupt?* we study three scenarios: (i) immediately after the failure; (ii) after a proactive checkpoint is taken and completed, and (iii) after the next regular checkpoint;

3. *when to steal a node?* we decide to actually steal a node only if (i) the victim job J_{victim} uses strictly less nodes than the failed job, or (ii) the victim job was released more recently than the failed job; or (iii) we compare the flow of the victim job and failed job in case we do interrupt the victim job or not, and retain the scenario leading to the smallest maximum flow for these two jobs.

Altogether, we implemented 18 variants of node stealing. All details on the implementation, the computation of the characteristics of the jobs for the re-execution of the failed and victim jobs, as well as the results of the simulations are available in Sections 4.6.1 to 4.6.3. In Section 4.6.1, we present additional variant heuristics. In Section 4.6.2, we propose the details on the implementation of other variant heuristics. In Section 4.6.3, we propose the results and discussion for the general heuristics. The global conclusion is the following: no variant improves the performance of SFSJ, which is sufficient to decrease the flow of large jobs, at the cost of a limited increase in the flow of small jobs. This corroborates the analysis of the utilization conducted in Section 4.5.1.

4.6.1 Design of node-stealing variants

The first question to deal with when studying node stealing is the choice of the victim job J_{victim}, that is, which job should be considered to be interrupted to free a node so that the failed job J_{failed} can be restarted. We consider here two possible choices:

V1 Select the currently running job with the smallest number of nodes as the victim job J_{victim}. If ties, choose the one whose release date is the latest (this is the solution chosen in Section 4.3.2 and evaluated in Section 4.4). The intuition for stopping small jobs is that they already have the smallest flows, and they are easy to reschedule.

V2 Select the currently running job with the latest release time as the victim job J_{victim}. If ties, choose the one whose number of nodes is smallest. The idea here is to stop jobs whose waiting times are among the smallest.

Once a victim is chosen, we need to decide when we will interrupt the victim job. We propose three scenarios for this timing decision:

T1 We immediately interrupt the victim job, and restart it from its previous checkpoint (this is the solution chosen in Section 4.3.2 and evaluated in Section 4.4).

T2 We proactively start a checkpoint on job J_{victim}, and stop this job right after the checkpoint. This avoids wasting computation time on J_{victim}, but induces some delay for the failed job J_{failed} as it can only be restarted after the checkpoint of J_{victim}.

T3 We wait for the next regular checkpoint of J_{victim}, and stop this job right after the checkpoint. This has a minimal impact on J_{victim} but induces a large waiting time for J_{failed}.

Finally, we need a criterion to decide it is worth interrupting it, or if we should rather wait for job to terminate to get a new node. We propose three criterion for this decision:

K1 If the victim job J_{victim} uses strictly less nodes than the failed job J_{failed}, we decide to interrupt J_{victim} (this is the decision taken in Section 4.3.2 and evaluated in Section 4.4).

K2 If the victim job J_{victim} was released more recently than failed job J_{failed}, we interrupt J_{victim}.

K3 We compute the flow of both jobs J_{victim} and J_{failed} based on their walltime in both scenarios (interrupting J_{victim} or waiting for a job completion), and we select the scenario that leads to the smallest maximum flow for these two jobs.

On the whole, we thus get 18 variants of node stealing, which are denoted by Hxyz, where x corresponds to timing choice Tx, y corresponds to victim choice Vy and z corresponds to interrupting choice Kz. For example SFSJ, the node stealing heuristic studied in Section 4.5, is denoted by H111.

4.6.2 Details on the implementation

In this section, we provide details on the implementation of the heuristics. We first give some insights on the implementation of the simulations, then we detail how to compute the remaining part of the victim job in the case of timing decision T2 and T3. We assume here that no new failure occurs until we have completely handled the current one, that is, until a checkpoint is taken and the failed job can be restarted. We finally explain how to handle the infrequent events of consecutive failures.

Simulation details

Algorithm 5 gives a precise statement of the various node stealing variants. Note that whenever a job is struck by a failure, its re-execution is submitted with priority 3. When a job is selected as a victim and interrupted, its re-execution is submitted with priority 2. Regular jobs have priority 1. In case of a failure (with or without node stealing), or in case of the rejuvenation of a node, the whole schedule is cleared and all jobs are rescheduled, by decreasing priority.

For timing variants T2 and T3 (proactive checkpointing and next checkpoint), the failed job is not resubmitted immediately after its failure. To avoid small jobs taking advantage of the nodes left idle by the failed jobs (that will be used for its re-execution), we submit a fictitious job to wait for the termination of the checkpoint on the victim job. If the failed job originally enrolled n nodes and had a single failure, this fictitious job uses $n - 1$ nodes. In case the failed job has no more remaining nodes (after one or multiple failures), then we can not submit this fictitious job. Because this fictitious job is needed to trigger the end of the proactive/future checkpoint on the victim job in our simulations, we cannot use node stealing in this situation. We thus cancel node stealing for this failed job and simply resubmit it from its last checkpoint. However, note that this concerns a very limited number of real scenarios.

Algorithm 5: General framework for heuristic $Hxyz$. Note that regular jobs have priority 1.

Input: Failed job J_{failed}, failure time t
if *there is an available node P at time t* **then**
    /* No need for node stealing */
    Submit the remaining part of J_{failed} (from its previous checkpoint) with priority 3 and allocate it to the previously allocated nodes that are still available and P
else
    Choose the victim J_{victim} according to victim choice Vy
    if *No victim is found or the interrupting criterion Kz is negative* **then**
        /* Do not use node stealing */
        Submit the remaining part of J_{failed} (from its previous checkpoint) with priority 3
    else
        /* Use node stealing */
        switch *Timing choice Tx* **do**
            case $T1$ **do**
                /* Interrupt victim right away */
                Interrupt victim job J_{victim}
                Submit the remaining part of J_{failed} with priority 3
                Submit the remaining part of J_{victim} with priority 2
            case $T2$ **do**
                /* Proactively checkpoint victim */
                Suspend job J_{victim} and initiate a checkpoint and wait for its completion
                Interrupt J_{victim}
                Submit the remaining part of J_{failed} with priority 3
                Submit the remaining part of J_{victim} with priority 2
            case $T3$ **do**
                /* Wait for next regular checkpoint */
                Wait for the completion of the next checkpoint of J_{victim}
                Interrupt J_{victim}
                Submit the remaining part of J_{failed} with priority 3
                Submit the remaining part of J_{victim} with priority 2

Computing the remaining part of the victim job

In all heuristics, the failed job is restarted from its previous checkpoint. The computation of the remaining part of the failed job has already been presented in Section 4.3.1. The job selected as the victim of the node stealing either (i) can be interrupted right away, as defined by timing decision T1 (in this case, the same formulas are used to compute the characteristics of its resubmission), or (ii) it can be interrupted later for timing decisions T2 and T3: we either proactively trigger a checkpoint, or wait for the next regular checkpoint. This requires to change the computation of the characteristics of the resubmitted victim job. We now details these computations.

Proactive checkpoint (timing decision T2) We consider here a victim job with a checkpoint period T, a checkpoint time C. In the proactive checkpoint, we may interrupt a job in the middle of a regular period, for example after a time $T_1 < T$ after the beginning of the period. Hence, when restarting the job, the first period may be different from the following ones, as it consists in completing the remaining part of this period, of length $T - T_1$. To deal with such cases, we denote by $T_{first} = T - T_1$ the duration of the first period. Since the job may be a resubmission of a previously failed or stopped job, we denote by R_{first} its initial recovery time. We have two cases:

- $R_{first} = 0$ in case of an initial submission,
- $R_{first} = R$ in case of a resubmission.

We consider that the victim job has a length t_{exec} and was started at t_{start}. The failure (on the failed job) happens at time t_{fail}. To simplify, we denote that $t_{run} = t_{fail} - t_{start}$ the length of the victim job up to the failure and $t_{first} = R_{first} + T_{first} + C$ the length of its first period, as it may differ from the following ones if the victim job is itself a resubmission of previously interrupted job. If the victim job is an initial submission, we just let $T_{first} = T$ and $R_{first} = 0$. In Figure 4.13, we illustrate these notations on the execution of a job. We will use times $t_1, \ldots, t_5$ as potential times for failures in the description of the formulas below.

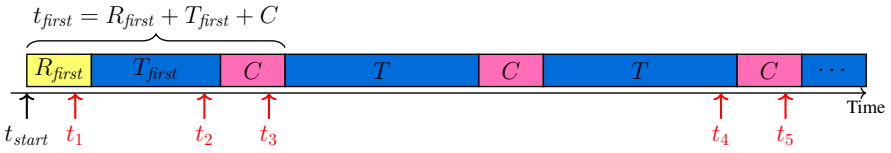


Figure 4.13: Illustration of the notations for the victim job, with the five cases distinguished to compute the proactive checkpoint strategy (T2).

We aim at computing when the victim job is interrupted and what are the characteristics of its resubmission. More precisely, we will first compute the following two quantities:

- The time t_{useful} spent by the victim job doing useful work until we stop it. This includes execution time and regular checkpoint time, but not the checkpoint that we introduce due to the proactive checkpoint strategy.
- The time $t_{checkpoint}$ need to complete the checkpoint introduced by the proactive checkpoint strategy. This checkpoint will be completed at time $t_{fail} + t_{checkpoint}$: at this time we will be able to steal a node from the victim to restart the failed job.

Then we will compute the characteristics of the resubmitted victim job: its length t'_{exec}, its first period length T'_{first} and its recovery time R'_{first}.

We distinguish five cases depending on which part of the job is struck by a failure.

The first case is when the failure occurs during the execution of the potential recovery R_{first}. This is when the failure hits at time $t_{fail} = t_1$ in the Figure 4.13. This means that no progress was made in the job ($t_{useful} = 0$), hence there is no need to start a proactive checkpoint and we can simply resubmit the job without any modifications. Note that this case only happens if the victim job was a re-execution of a job (as $R_{first} > 0$), so we have $R_{first} = R$, and $R'_{first} = R$,

The second case is when the failure occurs during the execution of the first period T_{first}, which corresponds to $t_{fail} = t_2$ in Figure 4.13. Then we start a proactive checkpoint at time t_{fail} to save the useful work executed in T_{first}. In this case, the saved useful work is $t_{useful} = t_{run} - R_{first}$ (since we classify regular checkpoint into the useful work, and proactive checkpoint not into the useful work). In the resubmission job, the first period will have to terminate the execution of the interrupted first period, that is, it will run from t_2 to T_{first}. Hence we will set $T'_{first} = T_{first} - (t_{run} - R_{first})$. The time needed to complete the proactive checkpoint is $t_{checkpoint} = C$.

The third case is when the failure occurs during the checkpoint that follows the first period, as illustrated by $t_{fail} = t_3$ in Figure 4.13. Then we do not need to start a proactive checkpoint, we simply wait for the completion of the ongoing checkpoint to be completed. The time we have to wait for the completion of the checkpoint is $t_{checkpoint} = t_{start} + t_{first} - t_{fail} = t_{first} - t_{run}$. In this case, the useful work performed by the job is $t_{useful} = T_{first} + C$ (containing the regular checkpoint into the useful work). The resubmission of the victim job will start by a regular period, that is, $T'_{first} = T$.

The fourth case happens when the failure occurs during a regular period T, which corresponds to $t_{fail} = t_4$ in Figure 4.13. We then start a proactive checkpoint at time t_{fail} to save the work already performed in this period, hence the duration of this checkpoint is $t_{checkpoint} = C$. The amount of successful work is thus $t_{useful} = t_{run} - R_{first}$. The first period of the resubmitted job copy will perform the missing work from t_{fail} to the end of the regular period, computed as

$$T'_{first} = T - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T + C} \right\rfloor \times (T + C)).$$

The fifth case happens when the failure occurs during a regular checkpoint, for example for $t_{fail} = t_5$ in Figure 4.13. As in the third case, we do not start a proactive checkpoint but take advantage of the ongoing one. In this case, the useful work starts at the beginning of T_{first} until the end of this regular checkpoint, that is

$$t_{useful} = T_{first} + C + \left\lceil \frac{t_{run} - t_{first}}{T + C} \right\rceil \times (T + C).$$

The time we have to wait until the end of the current checkpoint goes from t_{fail} until the end of the ongoing checkpoint, that is

$$t_{checkpoint} = T + C - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T + C} \right\rfloor \times (T + C)).$$

In this case, the first period of the resubmitted victim job is a regular one, so that $T'_{first} = T$.

In cases 2 to 5, we start proactive checkpoints, so that the recovery time for the resubmission of the victim job is set to $R'_{first} = R$.

Figure 4.14 summarizes how to compute the length of the resubmission of the victim job, as well as the time $t_{checkpoint}$ to wait until a node can be given to the failed job to restart it.

$$t_{useful} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ t_{run} - R_{first}, & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ T_{first} + C, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ t_{run} - R_{first}, & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T_{first} + C + \left\lceil \frac{t_{run} - t_{first}}{T+C} \right\rceil \times (T+C), & \text{otherwise.} \end{cases}$$

$$T'_{first} = \begin{cases} T_{first}, & \text{if } t_{run} \leq R_{first}, \\ T_{first} - (t_{run} - R_{first}), & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ T, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ T - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C)), & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T, & \text{otherwise.} \end{cases}$$

$$t_{checkpoint} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ C, & \text{if } R_{first} < t_{run} \leq R_{first} + T_{first}, \\ t_{first} - t_{run}, & \text{if } R_{first} + T_{first} < t_{run} \leq t_{first}, \\ C, & \text{if } t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C) \leq T, \\ T + C - (t_{run} - t_{first} - \left\lfloor \frac{t_{run} - t_{first}}{T+C} \right\rfloor \times (T+C)), & \text{otherwise.} \end{cases}$$

$$R'_{first} = R, \qquad t'_{exec} = R'_{first} + t_{exec} - t_{useful}, \qquad t'_{walltime} = R'_{first} + t_{walltime} - t_{useful}.$$

Figure 4.14: Formulas used to compute the useful executed work of the victim job t_{useful}, the length of the first period of the resubmitted victim job T'_{first}, the time between the failure and the completion of the proactive checkpoint $t_{checkpoint}$, the recovery time of the resubmitted victim job R'_{first}, and the length t'_{exec} and wall time $t'_{walltime}$ of the resubmitted victim job.

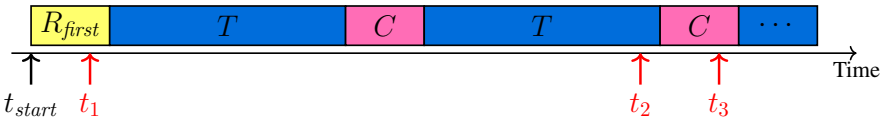


Figure 4.15: Illustration of the notations for the victim job, with the five cases distinguished to compute the future checkpoint strategy (T3).

Using next regular checkpoint (timing decision T3) We now detail how to compute the remaining time of the victim job as well as the time when a new node is available for the failed job in the case of the future checkpoint heuristic (timing decision T3). We use the definition introduced above for the proactive checkpoint heuristic. In the future checkpoint heuristic, all periods between checkpoints have the same length, contrarily to the proactive checkpoint heuristic when the first period may be different from the other. This means we always have $T_{first} = T$. This largely simplifies the analysis. We now distinguish between two cases depending on the state of the victim when the failure happens, illustrated on Figure 4.15.

The first case happens when the failure occurs during the execution of R_{first} by the victim job. This is the case for example for $t_{fail} = t_1$ in Figure 4.15. Then, there is no reason to wait for the next regular checkpoint, as no useful work has been performed by the victim job. We simply interrupt the victim and submit it again later. We thus have $t_{useful} = 0$ and $t_{checkpoint} = 0$.

The second case happens when the failure occurs after R_{first}, either during a regular execution or during a checkpoint, as illustrated by $t_{fail} = t_2$ or $t_{fail} = t_3$ on Figure 4.15. Then we need to wait until the next regular checkpoint of the victim job. In this case, the time performing useful work starts at the end of R_{first} and goes to the completion of the next checkpoint, that is, the one immediately following t_{fail} (we recall that regular checkpoints are counted as useful work). Hence we have

$$t_{useful} = \left\lceil \frac{t_{run} - R_{first}}{T + C} \right\rceil \times (T + C).$$

The time between the failure at t_{fail} and the completion of the next checkpoint can be computed as:

$$t_{checkpoint} = R_{first} + \left\lceil \frac{t_{run} - R_{first}}{T + C} \right\rceil \times (T + C) - t_{run}.$$

Again, the first case only happens when $R_{first} > 0$, that is $R_{first} = R$. Hence, in both cases, the recovery time of the victim job copy is $R'_{first} = R$.

To sum up, we compute the useful working time for the victim job as follows:

$$t_{useful} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T + C), & \text{otherwise.} \end{cases}$$

The time that we need to wait between t_{fail} and the completion of the next checkpoint of the victim job (which is the delay of the failed node needs to wait before being restarted with a stolen node) is computed as follows

$$t_{checkpoint} = \begin{cases} 0, & \text{if } t_{run} \leq R_{first}, \\ R_{first} + \left\lceil \frac{t_{run} - R_{first}}{T+C} \right\rceil \times (T + C) - t_{run}, & \text{otherwise.} \end{cases}$$

The length of the resubmission of the victim job is finally be computed as previously, as well as its wall time:

$$\begin{aligned} t'_{exec} &= R'_{first} + t_{exec} - t_{useful}, \\ t'_{walltime} &= R'_{first} + t_{walltime} - t_{useful}. \end{aligned}$$

Consecutive failures

In the previous discussion, we assumed that no failure hits either the victim job or the remaining node of the failed job until the checkpoint is completed and the failed job may be restarted. However, such rare cases can happen. We detail here how to handle them.

We assume that a job failed because of a failure at time t_{fail}^{first}. A victim job was selected in order to perform node stealing, that is, to relaunch the failed job with its remaining nodes plus one node of the victim job. In timing decision T2, we trigger a proactive checkpoint on the victim job at time t_{fail}^{first} as shown in Figure 4.16. In timing decision T3, we wait until the next regular checkpoint of the victim job, as shown in Figure 4.17. We now consider the event of a failure before the end of the checkpoint, on the victim job or on the nodes of the failed job that were not hit by a failure at time t_{fail}^{first} and remained available.

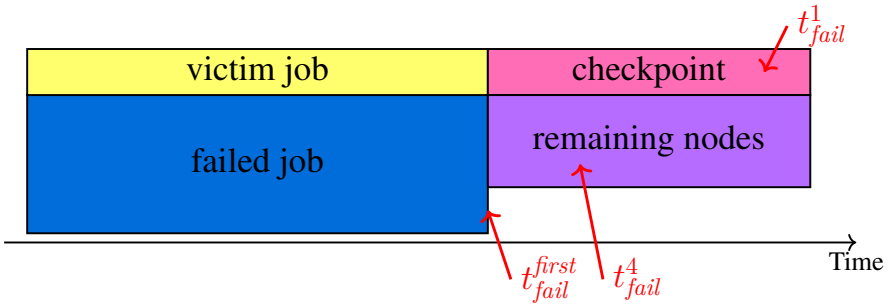


Figure 4.16: Notations of the special case for heuristic T2.

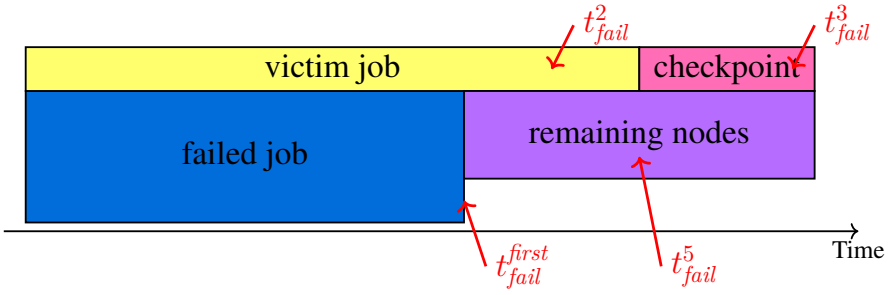


Figure 4.17: Notations of the special case for heuristic T3.

The first case is that old victim job or its proactive or regular checkpoint is hit by a new failure, as shown by t_{fail}^{1}, t_{fail}^{2} and t_{fail}^{3} in Figures 4.16 and 4.17. When this happens, we cancel node stealing for the originally failed job: both jobs are simply restarted from their previously successful checkpoints.

The second case deals with a new failure striking the remaining nodes of the originally failed job, as shown by t^4_{fail} and t^5_{fail} in Figures 4.16 and 4.17. In this case, there are two possibilites:

- The number of nodes of the victim job is smaller than the total number of failures that hit nodes of the failed job, which means the number of nodes of the old victim job is not sufficient to resume the old failed job. In this case, we also cancel node stealing and resubmit the failed job from its previous successful checkpoint. Since node stealing is canceled, the victim job continues its execution until its regular termination.
- The number of nodes of the victim job is larger than or equal to the number of failures occurred on nodes of the failed job, which means the number of nodes of the victim job is enough to resume the old failed job. In this case, we continue using node stealing for the failed job with the same victim job.

Another special case may occur when using future checkpoint (timing decision T3): the victim job may complete before its next checkpoint. In this case, we simply resubmit the failed job right after the termination of the victim job.

4.6.3 Presentation of all results and discussion

In this section, we report the results of experiments comparing the different variants of node stealing introduced above.

Utilization

We start by comparing the useful utilization of the platform by all heuristics, as presented in Table VIII which generalizes Table IV. This table also present the percentage of time at least one spare node is available (as previously in Table V). In this table (and below), we recall that variant xyz denotes the algorithm obtained with timing choice Tx, victim choice Vy and interrupting choice Kz.

We first remark that no variant is able to really increase the utilization above what is achieved by the initial node stealing heuristic (variant 111). Some of them even decrease the utilization below the one achieved by the baseline heuristic by up to 5%. As previously, we relate the impact on utilization to the percentage of time an idle node is available as a spare (and thus, node stealing is not useful). The table clearly shows that the larger this percentage of time, the smaller the impact of node stealing.

We also measure the number of time node stealing is used in Table IX. We remark that only variant 311, (corresponding to waiting the completion of the next regular checkpoint of the victim to interrupt it) is able to increase the usage of node stealing compared to our initial proposal. Using proactive checkpointing (variants $x**$) can (slightly) increase or decrease the use of node stealing. The other possibilities for y (choice of the victim) and z (interrupting criterion) always lead to a reduced usage of node stealing.

Job flows

Note that in the following figures (Figures 4.18 to 4.20), a few outliers has been omitted for better readability. There are described in Table X.

We first study the effect of changing the timing decision on the performance of node stealing. Figure 4.18 depicts the results when triggering proactive checkpoint, or when using the next regular checkpoint, rather than interrupting the victim as soon as possible. We notice that no strategy is able to clearly outperform the original node stealing heuristic. Waiting for the next checkpoint always increases

Variant	June 2017		March 2018	
	utilization	idle perc.	utilization	idle perc.
baseline	79.32 %	92.09 %	77.39 %	93.36 %
111	80.69 %	90.67 %	78.50 %	92.48 %
112	80.15 %	91.95 %	78.12 %	92.00 %
113	80.49 %	91.85 %	77.64 %	92.65 %
121	80.48 %	91.67 %	77.47 %	92.61 %
122	79.40 %	91.50 %	76.95 %	92.63 %
123	79.20 %	91.76 %	77.15 %	92.77 %
211	80.52 %	90.87 %	78.09 %	92.01 %
212	80.30 %	90.81 %	78.11 %	92.53 %
213	80.50 %	90.35 %	77.91 %	92.18 %
221	80.50 %	91.16 %	78.14 %	92.45 %
222	79.60 %	91.22 %	77.94 %	92.46 %
223	80.17 %	91.06 %	78.31 %	92.42 %
311	76.18 %	85.87 %	75.17 %	88.90 %
312	77.50 %	89.48 %	76.41 %	90.57 %
313	78.35 %	90.32 %	76.87 %	91.38 %
321	78.82 %	89.87 %	76.88 %	92.05 %
322	78.57 %	89.03 %	75.92 %	90.09 %
323	78.57 %	89.68 %	77.28 %	92.15 %

Table VIII: Utilization and percentage of time at least one node is available (idle perc.) in all variants. Results are for the Mira platform in June 2017 and March 2018.

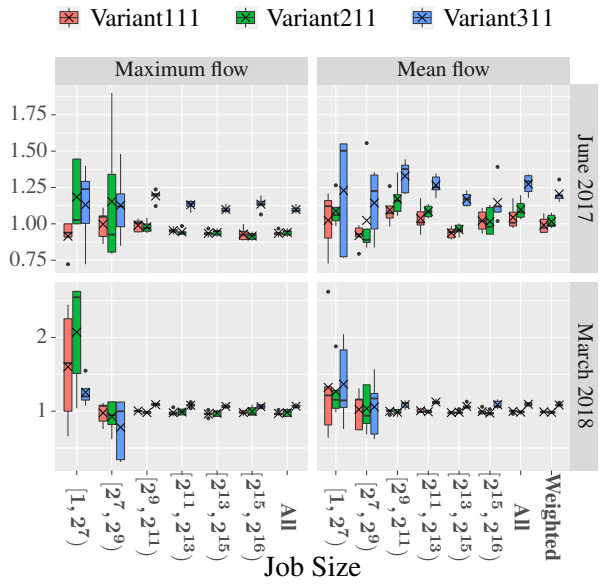


Figure 4.18: Maximum flow and mean flow relative to BASELINE for various timing decision (111: immediately interrupting, 211: proactive checkpointing, 311: waiting next checkpoint) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

Variant	June 2017			March 2018		
	node stealing	empty node	total failures	node stealing	empty node	total failures
111	63.4	404.4	467.8	46.6	416.2	462.8
112	33.6	433.4	467.0	44.6	415.6	460.2
113	48.0	423.6	471.6	44.6	420.0	464.6
121	32.4	440.4	472.8	29.8	432.6	462.4
122	46.8	422.4	469.2	49.8	414.4	464.2
123	48.0	418.2	466.2	48.2	414.2	462.4
211	60.8	413.2	474.0	53.2	411.2	464.4
212	34.6	431.6	466.2	43.8	417.2	461.0
213	47.6	424.2	471.8	41.0	419.8	460.8
221	33.2	432.2	465.4	27.4	432.4	459.8
222	49.8	410.4	460.2	48.0	413.2	461.2
223	51.4	417.6	469.0	49.4	411.8	461.2
311	76.4	362.6	439.0	60.2	383.6	443.8
312	39.8	413.4	453.2	43.6	408.4	452.0
313	17.2	437.4	454.6	14.8	441.2	456.0
321	32.8	427.4	460.2	26.2	430.4	456.6
322	55.8	395.8	451.6	57.0	394.0	451.0
323	30.6	429.4	460.0	24.0	435.4	459.4

Table IX: Number of time node stealing is used vs number of time an empty node is used for all node stealing variants. Results are for the Mira platform in June 2017 and March 2018.

the maximum and average flows. Using proactive checkpointing has comparable performance with the original node stealing for large jobs, but sometimes largely increases the maximum flow of small jobs.

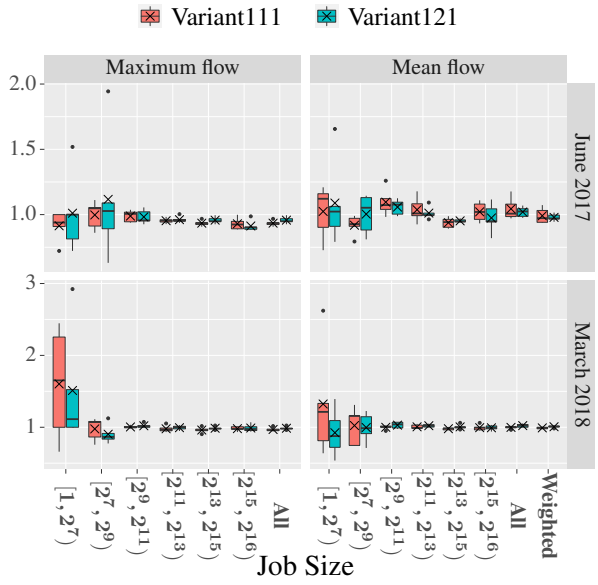


Figure 4.19: Maximum flow and mean flow relative to BASELINE heuristic for the two victim choices (111: fewest nodes, 121: latest release date) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

We then study on Figure 4.19 the impact of the choice of the victim on the performance of node stealing. In the original node stealing, we select the job with the smallest number of nodes. In the proposed variant, we select the job with the latest release date. We see that the victim selection policy has a limited impact for small jobs, but very little for large jobs. On the whole, it does not allow to improve performance.

Finally, Figure 4.20 presents the results when changing the interrupting criterion. In the original heuristic, we decide to interrupt a victim job and perform node stealing if the victim requires less nodes than the failed job. We also proposed to use release date to take this decision, by interrupting a victim only if it was release later than the failed job. The last criterion requires to compute an estimation of the flow for both the failed and victim job: the victim is interrupted only if it leads to a smaller maximum flow for both jobs. We notice in these results that changing the interrupting criterion has an impact only on small jobs, and does not clearly improve the results.

On the whole, all proposed variants fail to clearly improve the performance of node stealing: the basic node stealing heuristic is sufficient to improve the flow of large jobs, at the cost of a limited increase in the flow of small jobs (which is originally much smaller than the one of large jobs).

4.7 Conclusion

Patel et al wrote in their SC'20 paper [82]: *Users are now submitting medium-sized jobs because the waits times for larger sizes tends to be longer*. Indeed, we have shown that failures dramatically increase the flow of large jobs. It is important to invent scheduling strategies that decrease the flow of large jobs on large-scale machines.

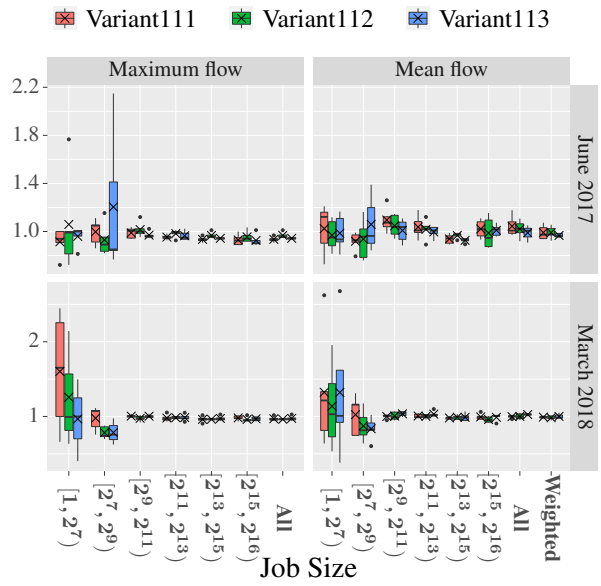


Figure 4.20: Maximum flow and mean flow relative to BASELINE for the various interrupting criterion (111: less nodes, 112: later release date, 113: better estimated maximum flow) and for various categories of job sizes. MTBF and downtime are both set to 1 hour.

	Variant	job size	value	type of figure
Figure 4.18	111	$[1, 2^7)$	7.99	Maxflow
	311	$[1, 2^7)$	6.09	Maxflow
Figure 4.19	111	$[1, 2^7)$	7.99	Maxflow
Figure 4.20	111	$[1, 2^7)$	7.99	Maxflow
	112	$[1, 2^7)$	5.47	Maxflow
	113	$[1, 2^7)$	4.09	Maxflow
	113	$[1, 2^7)$	4.84	Maxflow
	112	$[1, 2^7)$	7.73	Maxflow
	112	$[1, 2^7)$	3.58	Meanflow

Table X: Outliers removed from Figures 4.18 to 4.20 for the March 2018 dataset.

We have introduced node stealing as an efficient approach to decrease the flow of large jobs. For example, in June 2017 on Mira, the maximum flow of large jobs ($[32K, 64K)$ nodes) goes down from 7.20 to 3.72 days, while the maximum flow of small jobs ($[1, 128)$ nodes) increases from 0.19 to 0.54 days. We argue that the sharp decrease of the flow of large jobs is well worth the small increase of the flow of small jobs, given that large-scale platforms are primarily intended to execute large jobs. A side advantage of node stealing is a slight increase in terms of platform utilization. We have designed several variants of node stealing and report that they behave similarly.

This study opens interesting problems in this area, such as designing a node-stealing-aware batch scheduler: when taking scheduling decisions at submission time, the goal would be to account for the possibility of mitigating a failure by node stealing.

Currently node stealing is used very few times because there is often a node available when a failure strikes a job. Hence the failed job could restart with this node instead of requiring to steal a node from another running job. This opportunity would be drastically limited on architectures where topology matters: even if a spare node is available, it may not be possible to use it because of topology constraints. In such a framework, node-stealing would probably lead to a dramatic decrease in response time.

Conclusion

In this thesis, we have studied resilience techniques to deal with future Exascale platforms. More precisely, we designed new scheduling strategies to account for various objectives, such as minimizing makespan, maximizing utilization of the platform, and minimizing job flows. Chapter 2 and Chapter 3 of this thesis focused on designing fault-tolerance algorithms for iterative applications, while Chapter 4 focused on introducing scheduling heuristics for batch schedulers.

Chapter 2 and Chapter 3 both dealt with iterative applications, but there are some differences in the application model. In Chapter 2, we dealt with linear chains whose tasks do not have constant execution times but obey some probability distributions, while in Chapter 3, we considered the iterative application as a chain of cyclic tasks. The goal of both chapters is to minimize makespan for the iterative applications. The heuristics proposed in these two chapters are compared with the traditional Young/Daly formula. Chapter 2 shows Young/Daly formula can be safely applied to stochastic iteration applications, and Chapter 3 shows Young/Daly formula is suboptimal compared to our optimal algorithm proposed for the deterministic iterative applications. In Chapter 4, we considered how to schedule a failed job that was executing on an over-subscribed platform in order to maximize the utilization of the platform and minimize the flows of the larger jobs. We proposed a novel approach called node stealing which interrupts another currently executing job, steals one of its nodes, and assigns it as a new resource to the failed job.

The main contributions are summarized in the following three paragraphs:

Stochastic iterative applications (Chapter 2)

In this chapter, we introduced and analyzed checkpointing strategies for stochastic iterative applications whose execution times of iterations are modeled with probabilistic distributions. We first proposed the static strategy where checkpoints are taken after a given number of iterations and provided a closed-form formula to compute the optimal period for any distribution. Then we proposed the dynamic strategy where checkpoints are taken only if the total amount of work since the last checkpoint exceeds a given threshold which can be computed by a closed-form formula we provided. We showed that the first-order approximations of both formulas correspond to the Young/Daly formula. Extensive simulations showed our mathematical formula fits the evaluated execution time and the makespan obtained by the Young/Daly formula is always within one percent of the makespan obtained by the optimal strategy. Thus, the Young/Daly formula can be applied safely to the checkpoint strategy of stochastic iterative applications.

Deterministic iterative applications (Chapter 3)

In this chapter, we investigated checkpointing strategies for deterministic iterative applications. Each iteration is composed of a chain of tasks, and these tasks have different lengths and different checkpoint costs. We showed that there exists an optimal periodic strategy that a dynamic-programming algorithm

can compute. We conducted simulations with both synthetic and real-life workflows to compare our optimal algorithm with four natural competitor heuristics: The first two checkpoints after each task or each iteration, respectively. The last two are extensions of the Young/Daly formula to iterative applications. Simulation results showed that our optimal algorithm could reduce the makespan of iterative applications for all problem instances.

Node stealing for failed jobs (Chapter 4)

In this chapter, we first showed that failures dramatically increase the flow of large jobs. It is important to invent scheduling strategies that decrease the flow of large jobs on large-scale platforms. In order to decrease the flow of large jobs, we introduced an efficient approach called node stealing which interrupts another currently executing job, steals one of its nodes, and assigns it as a new resource to the failed job. We first focused on SFSJ (*Steal From Small Jobs*), a strategy which chooses the job to interrupt among those with the smallest number of nodes and, if ties, with the shortest execution time so far. Simulations with both real-life and synthetic workflows showed that SFSJ could improve the utilization of the platform and sharply decrease the flow of large jobs at the cost of slightly increasing the flow of small jobs. Then we designed and implemented several variants of node stealing by considering which job to interrupt, when to interrupt and when to steal a node. All variants behaved similarly and the primary node stealing heuristic SFSJ is sufficient.

Perspectives

We intend to continue our research along the following short-term and long-term perspectives.

Short-term perspectives

In the short term, we plan to improve the limitations of our current work and extend our solution to more general situations. We listed the following points:

- In the work of Chapter 2, we focused on single-level checkpointing. Extending our approach to multi-level checkpointing protocols would be an interesting topic. Multi-level checkpointing protocols are state-of-the-art approaches in which checkpoints are taken at different periods for each level of fault. Intuitively, the fewer failures, the longer the checkpoint period. Even in a deterministic setting, it is hard to model and optimize analytically. Such an extension to a stochastic framework is quite challenging.

- In the study of Chapter 3, we were able to deal with iterative applications whose iterations are composed of a linear chain. Perspectives will be devoted to dealing with iterative applications whose iterations are composed of a Directed Acyclic Graph (DAG) of tasks. Such an extension poses another major challenge that several tasks may execute concurrently on the platform. Thus, we will likely only obtain suboptimal algorithms instead of optimal ones.

- In Chapter 4, we proposed a node stealing approach based on the classic conservative backfilling strategy used by batch schedulers. One of the backfilling strategies is conservative backfilling, which means a backfilled job can delay no job in the queue. Another flavor is EASY, which means backfilled jobs never delay the first job in the queue. Perspectives will be devoted to exploring other well-established batch scheduling strategies (such as EASY) and assessing the usefulness of node stealing when coupled with these strategies.

- It would also be practical to do simulations on a real machine to test the effectiveness of our algorithms. Nevertheless, controlling the probability of failures to simulate different scenarios is a challenging problem.

Long-term perspectives: ABFT for iterative applications

ABFT (Algorithm based on fault-tolerance) is proposed to detect soft errors based on the specific mathematical properties of different iterative methods. If the Krylov subspace method is not restarted after too many iterations, the subspace dimension and the number of basis vectors will become too large. Then the basis vectors will lose orthogonality due to rounding error, which will lead to an inaccurate calculation result. Thus, it usually checks if the subspace basis vectors satisfy the orthogonality relation. As stated in Section 1.3, some traditional Krylov subspace methods such as CG and GMRES have already been designed ABFT variants for their specific mathematical properties. Kaczmarz-type inner-iteration preconditioned flexible GMRES method is the state-of-the-art Krylov subspace method [32]. This method is an inner-outer iterative method, which uses the Flexible GMRES method as the outer iteration and the randomized/greedy Kaczmarz method as the inner iteration. Since it uses the Flexible framework, we can naturally apply the existing ABFT variant designed for GMRES to it, but will there be some new results based on the different mathematical properties of the new method? In addition, the new iterative application can be modeled as a chain of tasks, and each task of the linear chain has different lengths and checkpoint costs. Can we design similar fault tolerance algorithms like Chapter 3?

Long-term perspectives: node-stealing-aware batch scheduler

In Chapter 4, we came up with the idea of node stealing and proposed variants considering different design choices such as which job to choose as the victim job, when to interrupt the victim job and when to steal a node. From the simulation results, it can be observed that the number of activating node-stealing is only 10% of the number of total failures. Because of the topology of the architecture, it may be impossible to steal a specific node even if it is idle. If the number of times that node-stealing activates increases, the scheduling heuristic may be more effective. In addition, the node-stealing heuristic can significantly decrease the flow of large jobs at the expense of increasing the flow of small jobs. Although it might be acceptable because the magnitude of the increase or decrease is not the same, can we design a node-stealing-aware batch scheduler to account for the possibility of mitigating a failure by node stealing when taking scheduling decisions at submission time? It may result in a better balance between large and small jobs to get a more significant overall benefit.

Bibliography

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. "Adaptive Incremental Checkpointing for Massively Parallel Systems." In: *18th Int. Conf. Supercomputing*. ACM, 2004.

[2] E. Agullo, M. Altenbernd, H. Anzt, L. Bautista-Gomez, T. Benacchio, L. Bonaventura, H.-J. Bungartz, S. Chatterjee, F. M. Ciorba, N. DeBardeleben, et al. "Resiliency in numerical algorithm design for extreme scale simulations." In: *The International Journal of High Performance Computing Applications* 36.2 (2021), pp. 251–285.

[3] E. Agullo, S. Cools, E. F. Yetkin, L. Giraud, N. Schenkels, and W. Vanroose. "On soft errors in the conjugate gradient method: sensitivity and robust numerical detection." In: *SIAM Journal on Scientific Computing* 42.6 (2020), pp. C335–C358.

[4] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon. "Numerical recovery strategies for parallel resilient Krylov linear solvers." In: *Numerical Linear Algebra with Applications* 23.5 (2016), pp. 888–905.

[5] E. Agullo, L. Giraud, P. Salas, and M. Zounon. "Interpolation-restart strategies for resilient eigensolvers." In: *SIAM Journal on Scientific Computing* 38.5 (2016), pp. C560–C583.

[6] E. Agullo, L. Giraud, and M. Zounon. "On the resilience of parallel sparse hybrid solvers." In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE. 2015, pp. 75–84.

[7] *APEX Worflows*. Tech. rep. Available online at http://www.nersc.gov/assets/apex-workflows-v2.pdf. Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratory (SNL)., 2016.

[8] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. "Checkpointing strategies for scheduling computational workflows." In: *Int. J. Networking and Computing* 6.1 (2016), pp. 2–26.

[9] Z. Bai and W. Wu. "On greedy randomized coordinate descent methods for solving large linear least-squares problems." In: *Numer. Linear Algebr. Appl.* 26.4 (2019), pp. 1–15.

[10] Z. Bai and W. Wu. "On greedy randomized Kaczmarz method for solving large sparse linear systems." In: *SIAM J. Sci. Comput.* 40.1 (2018), A592–A606.

[11] T. Benacchio, L. Bonaventura, M. Altenbernd, C. D. Cantwell, P. D. Düben, M. Gillard, L. Giraud, D. Göddeke, E. Raffin, K. Teranishi, et al. "Resilience and fault tolerance in high-performance computing for numerical weather and climate prediction." In: *The International Journal of High Performance Computing Applications* 35.4 (2021), pp. 285–311.

[12] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. "Towards optimal multi-level checkpointing." In: *IEEE Trans. Computers* 66.7 (2017), pp. 1212–1226.

[13] V. Bharadwaj, D. Ghose, and T. Robertazzi. "Divisible load theory: a new paradigm for load scheduling in distributed systems." In: *Cluster Computing* 6.1 (2003), pp. 7–17.

[14] R. H. Bisseling and A.-J. N. Yzelman. "Thinking in Sync: The Bulk-Synchronous Parallel Approach to Large-Scale Computing." In: *ACM Computing reviews* 57.6 (2016), pp. 322–327.

[15] D. Blackston and T. Suel. "Highly portable and efficient implementations of parallel adaptive n-body methods." In: *Proc. ACM Supercomputing*. 1997, pp. 1–20.

[16] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. *Checkpointing strategies for parallel jobs*. Research Report 7520. Short version appears in SC'2011. France: INRIA, Jan. 2011.

[17] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. "Fault-tolerant linear solvers via selective reliability." In: *arXiv preprint arXiv:1206.1390* (2012).

[18] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. "A batch scheduler with high level components." In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* Vol. 2. IEEE. 2005, pp. 776–783.

[19] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience." In: *Int. J. High Performance Computing Applications* 23.4 (2009), pp. 374–388.

[20] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014).

[21] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. "Versatile, scalable, and accurate simulation of distributed applications and platforms." In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2899–2917.

[22] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. "Fault resilience of the algebraic multi-grid solver." In: *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, pp. 91–100.

[23] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." In: *ACM Transactions on Computer Systems* 3.1 (1985), pp. 63–75.

[24] C. Chekuri, W. Hasan, and R. Motwani. "Scheduling problems in parallel query optimization." In: *PODS'1995, the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. San Jose, California, United States: ACM Press, 1995, pp. 255–265. ISBN: 0-89791-730-8. DOI: http://doi.acm.org/10.1145/212433.212471.

[25] Z. Chen. "Algorithm-based recovery for iterative methods without checkpointing." In: *Proceedings of the 20th international symposium on High performance distributed computing*. 2011, pp. 73–84.

[26] Z. Chen. "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods." In: *ACM SIGPLAN Notices* 48.8 (2013), pp. 167–176.

[27] A. Choudhary, W.-k. Liao, D. Weiner, P. Varshney, R. Linderman, M. Linderman, and R. Brown. "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers." In: *IEEE Transactions on Aerospace and Electronic Systems* 36.2 (Apr. 2000), pp. 655–662. DOI: 10.1109/7.845238.

[28] S. Coghlan, K. Kumaran, R. M. Loy, P. Messina, V. Morozov, J. C. Osborn, S. Parker, K. Riley, N. A. Romero, and T. J. Williams. "Argonne applications for the IBM Blue Gene/Q, Mira." In: *IBM Journal of Research and Development* 57.1/2 (2013), pp. 12–1.

[29] J. T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps." In: *Future Generation Comp. Syst.* 22.3 (2006), pp. 303–312.

[30] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. "Grid Resource Management." In: Springer, 2003. Chap. Workflow management in GriPhyN.

[31] 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.

[32] Y. Du, K. Hayami, N. Zheng, K. Morikuni, and J. Yin. “Kaczmarz-type inner-iteration preconditioned flexible GMRES methods for consistent linear systems.” In: *SIAM Journal on Scientific Computing* (2021), S345–S366.

[33] Y. Du, L. Marchal, G. Pallez, and Y. Robert. *Code for simulations of deterministic iterative applications*. https://github.com/Yishu0604/Optimal-Checkpointing-Strategies-for-Iterative-Applications.

[34] Y. Du, L. Marchal, G. Pallez, and Y. Robert. *Code for simulations of stochastic iterative applications*. https://gitlab.inria.fr/yisdu/yd-segmented-simulation.

[35] Y. Du, L. Marchal, G. Pallez, and Y. Robert. *Implementation of the node stealing scheduler in Batsched*. https://gitlab.inria.fr/yishu/node-stealing-for-resilience/-/tree/master/tmp.

[36] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard. “Batsim: a realistic language-independent resources and jobs management systems simulator.” In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2015, pp. 178–197.

[37] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. “Variational iterative methods for nonsymmetric systems of linear equations.” In: *SIAM Journal on Numerical Analysis* 20.2 (1983), pp. 345–357.

[38] J. Elliott, M. Hoemmen, and F. Mueller. “Evaluating the impact of SDC on the GMRES iterative solver.” In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1193–1202.

[39] 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.

[40] R. Fletcher. “Conjugate gradient methods for indefinite systems.” In: *Numer. Anal.* 506 (1976), pp. 73–89.

[41] S. P. Frankel. “Convergence rates of iterative treatments of partial differential equations.” In: *Mathematical Tables and Other Aids to Computation* 4.30 (1950), pp. 65–75.

[42] R. W. Freud and N. M. Nachtigal. “QMR: a quasi-minimal residual method for non-Hermitian linear systems.” In: *Numer. Math.* 60 (1991), pp. 315–339.

[43] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez. “Profiles of upcoming HPC Applications and their Impact on Reservation Strategies.” In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2020), pp. 1178–1190.

[44] A. V. Gerbessiotis and L. G. Valiant. “Direct bulk-synchronous parallel algorithms.” In: *J. Parallel Distributed Computing* 22.2 (1994), pp. 251–267.

[45] G. H. Golub and H. A. Van Der Vorst. “Closer to the solutions: iterative linear solvers.” In: *The state of the art in numerical analysis* (1997), pp. 63–92.

[46] R. M. Gower and P. Richtárik. “Randomized Iterative Methods for Linear Systems.” In: *SIAM Journal on Matrix Analysis and Applications* 36.4 (2015), pp. 1660–1690.

[47] F. Guirado, A. Ripoll, C. Roig, and E. Luque. “Optimizing Latency under Throughput Requirements for Streaming Applications on Cluster Execution.” In: *Cluster Computing, 2005. IEEE International*. Sept. 2005, pp. 1–10. DOI: 10.1109/CLUSTR.2005.347051.

[48] F. Guirado, A. Ripoll, C. Roig, A. Hernandez, and E. Luque. “Exploiting Throughput for Pipeline Execution in Streaming Image Processing Applications.” In: *Euro-Par 2006, Parallel Processing*. LNCS 4128. Springer, 2006, pp. 1095–1105.

[49] M. H. Gutknecht. “Variants of BICGSTAB for matrices with complex spectrum.” In: *SIAM J. Scientific Computing* 14.5 (1993), pp. 1020–1033.

[50] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien. “Checkpointing workflows for fail-stop errors.” In: *IEEE Trans. Computers* 67.8 (2018), pp. 1105–1120.

[51] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. “A Generic Approach to Scheduling and Checkpointing Workflows.” In: *Int. Journal of High Performance Computing Applications* 33.6 (2019), pp. 1255–1274.

[52] P. Hanrahan, D. Salzman, and L. Aupperle. “A rapid hierarchical radiosity algorithm.” In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. 1991, pp. 197–206.

[53] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. “Size-based scheduling to improve web performance.” In: *ACM Transactions on Computer Systems (TOCS)* 21.2 (2003), pp. 207–233.

[54] T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Ozguner, and Ü. V. Çatalyürek. “Investigating the Use of GPU-Accelerated Nodes for SAR Image Formation.” In: *Proceedings of the IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*. 2009.

[55] T. Herault and Y. Robert, eds. *Fault-Tolerance Techniques for High-Performance Computing*. Computer Communications and Networks. Springer Verlag, 2015.

[56] M. R. Hestenes and E. L. Stiefel. “Methods of conjugate gradients for solving linear systems.” In: *J. Res. Nat. Bur. Standards* 49 (1952), pp. 409–436.

[57] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. “BSPlib: The BSP programming library.” In: *Parallel Computing* 24.14 (1998), pp. 1947–1980.

[58] A. Hori, K. Yoshinaga, T. Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa. “Sliding Substitution of Failed Nodes.” In: *Proceedings of the 22nd European MPI Users’ Group Meeting*. EuroMPI ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450337953. DOI: 10.1145/2802658.2802670. URL: https://doi.org/10.1145/2802658.2802670.

[59] Y. Huo, Z. Xu, Y. Xiong, K. Aboud, P. Parvathaneni, S. Bao, C. Bermudez, S. M. Resnick, L. E. Cutting, and B. A. Landman. “3D whole brain segmentation using spatially localized atlas network tiles.” In: *NeuroImage* 194 (2019), pp. 105–119.

[60] D. Jackson, Q. Snell, and M. Clement. “Core algorithms of the Maui scheduler.” In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2001, pp. 87–102.

[61] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero. “Exploiting asynchrony from exact forward recovery for due in iterative solvers.” In: *SC’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2015, pp. 1–12.

[62] L. Jaulmes, M. Moreto, E. Ayguade, J. Labarta, M. Valero, and M. Casas. “Asynchronous and exact forward recovery for detected errors in iterative solvers.” In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 1961–1974.

[63] J. Kim, Y. Gil, and M. Spraragen. “A knowledge-based approach to interactive workflow composition.” In: *14th International Conference on Automatic Planning and Scheduling (ICAPS 04)*. 2004.

[64] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. “Scheduling constrained dynamic applications on clusters.” In: *Supercomputing’1999, the 1999 ACM/IEEE conference on Supercomputing*. Portland, Oregon, United States: ACM, 1999, p. 46. ISBN: 1-58113-091-0. DOI: http://doi.acm.org/10.1145/331532.331578.

[65] D. Kumar, G. Baranwal, Z. Raza, and D. P. Vidyarthi. “A survey on spot pricing in cloud computing.” In: *Journal of Network and Systems Management* 26.4 (2018), pp. 809–856.

[66] C. Lanczos. “Solution of systems of linear equations by minimized iterations.” In: *J. Res. Nat. Bur. Standards* 49 (1952), pp. 33–53.

[67] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. “Recovery patterns for iterative methods in a parallel unstable environment.” In: *SIAM J.Scientific Computing* 30.1 (2008), pp. 102–116.

[68] D. Leventhal and A. S. Lewis. “Randomized methods for linear constraints: convergence rates and conditioning.” In: *Math. Operations Research* 35.3 (2010), pp. 641–654.

[69] L. Lin, L. Pan, and S. Liu. “Backup or not: An online cost optimal algorithm for data analysis jobs using spot instances.” In: *IEEE Access* 8 (2020), pp. 144945–144956.

[70] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, et al. “Top ten exascale research challenges.” In: *DOE ASCAC subcommittee report* (2014), pp. 1–86.

[71] A. Ma, D. Needell, and A. Ramdas. “Convergence properties of the randomized extended Gauss–Seidel and Kaczmarz methods.” In: *SIAM Journal on Matrix Analysis and Applications* 36.4 (2015), pp. 1590–1604.

[72] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System.” In: *SC*. ACM, 2010.

[73] A. W. Mu’alem and D. G. Feitelson. “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling.” In: *IEEE transactions on parallel and distributed systems* 12.6 (2001), pp. 529–543.

[74] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. “Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments.” In: *Eighth IEEE Int. Symp. Cluster Computing and the Grid (CCGRID)*. 2008.

[75] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello. “VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale.” In: *Int. Parallel Distr. Processing Symp. (IPDPS)*. IEEE, 2019.

[76] W. L. Oberkampf and C. J. Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[77] M. E. Ozturk, G. Agrawal, Y. Li, and C.-S. Chou. *Handling Soft Errors in Krylov Subspace Methods by Exploiting Their Numerical Properties*. Tech. rep. EasyChair, 2020.

[78] C. Pachajoa, M. Levonyak, and W. N. Gansterer. “Extending and Evaluating Fault-Tolerant Preconditioned Conjugate Gradient Methods.” In: *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2018, pp. 49–58.

[79] C. Pachajoa, M. Levonyak, W. N. Gansterer, and J. L. Träff. "How to make the preconditioned conjugate gradient method resilient against multiple node failures." In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[80] C. Pachajoa, C. Pacher, M. Levonyak, and W. N. Gansterer. "Algorithm-Based Checkpoint-Recovery for the Conjugate Gradient Method." In: *49th International Conference on Parallel Processing-ICPP*. 2020, pp. 1–11.

[81] C. C. Paige and M. A. Saunders. "Solution of sparse indefinite systems of linear equations." In: *SIAM J. Numer. Anal.* 12 (1975), pp. 617–629.

[82] 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*. IEEE. 2020, pp. 1–17.

[83] D. Petcu. "The performance of parallel iterative solvers." In: *Computers and Mathematics with Applications* 50.7 (2005), pp. 1179–1189.

[84] S. Prabhakaran, M. Neumann, and F. Wolf. "Efficient fault tolerance through dynamic node replacement." In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2018, pp. 163–172.

[85] T. Robertazzi. "Ten reasons to use Divisible Load Theory." In: *IEEE Computer* 36.5 (2003), pp. 63–68.

[86] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo. "The discovery net system for high throughput bioinformatics." In: *Bioinformatics* 19.Suppl 1 (2003), pp. i225–31.

[87] C. J. Roy and W. L. Oberkampf. "A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing." In: *Computer methods in applied mechanics and engineering* 200.25-28 (2011), pp. 2131–2144.

[88] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. Society for Industrial and Applied Mathematics, 2003.

[89] Y. Saad and M. H. Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems." In: *SIAM J. Scientific Statistical Computing* 7.3 (1986), pp. 856–869.

[90] P. Sao and R. Vuduc. "Self-stabilizing iterative solvers." In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2013, pp. 1–8.

[91] I. S. L. J. Scheduler. *Fault tolerance and automatic management host failover*. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=cluster-fault-tolerance. 2021.

[92] S. Schlagkamp, R. Ferreira da Silva, W. Allcock, E. Deelman, and U. Schwiegelshohn. "Consecutive job submission behavior at Mira supercomputer." In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 2016, pp. 93–96.

[93] O. Sertel, J. Kong, H. Shimada, Ü. V. Çatalyürek, J. H. Saltz, and M. N. Gurcan. "Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development." In: *Pattern Recognition* 42.6 (2009), pp. 1093–1103.

[94] R. F. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. "A characterization of workflow management systems for extreme-scale applications." In: *Future Generation Computer Systems* 75 (2017), pp. 228–238.

[95] V. Simoncini. "On the convergence of restarted Krylov subspace methods." In: *SIAM Journal on Matrix Analysis and Applications* 22.2 (2000), pp. 430–452.

[96] Slurm team. *Slurm Multifactor Priority Plugin*. https://slurm.schedmd.com/priority_multifactor.html. 2020.

[97] Slurm team. *Slurm Workload Manager*. https://slurm.schedmd.com/. 2020.

[98] P. Sonneveld. "CGS, a fast Lanczos-type solver for nonsymmetric linear systems." In: *SIAM J. Sci. Statist. Comput.* 10 (1989), pp. 36–52.

[99] T. Strohmer and R. Vershynin. "A randomized Kaczmarz algorithm with exponential convergence." In: *J. Fourier Analysis and Applications* 15.2 (2009), p. 262.

[100] W. Tang, N. Desai, D. Buettner, and Z. Lan. "Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P." In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–11.

[101] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello. "Improving performance of iterative methods by lossy checkponting." In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 52–65.

[102] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen. "New-sum: A novel online abft scheme for general iterative methods." In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 2016, pp. 43–55.

[103] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, et al. *Workflows for e-Science: scientific workflows for grids*. Vol. 1. Springer, 2007.

[104] Top500. *Top 500 Supercomputer Sites*. https://www.top500.org/lists/top500/2022/06/. June 2022.

[105] S. Toueg and Ö. Babaoğlu. "On the Optimum Checkpoint Selection Problem." In: *SIAM J. Comput.* 13.3 (1984).

[106] L. G. Valiant. "The Complexity of Enumeration and Reliability Problems." In: *SIAM J. Comput.* 8.3 (1979), pp. 410–421.

[107] K. Yamamoto, A. Uno, H. Murai, T. Tsukamoto, F. Shoji, S. Matsui, R. Sekizawa, F. Sueyasu, H. Uchiyama, M. Okamoto, N. Ohgushi, K. Takashina, D. Wakabayashi, Y. Taguchi, and M. Yokokawa. "The K computer Operations: Experiences and Statistics." In: *Proceedings of the International Conference on Computational Science (ICCS)*. Ed. by D. Abramson, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot. Vol. 29. Procedia Computer Science. Elsevier, 2014, pp. 576–585. DOI: 10.1016/j.procs.2014.05.052. URL: https://doi.org/10.1016/j.procs.2014.05.052.

[108] D. Young. "Iterative methods for solving partial difference equations of elliptic type." In: *Transactions of the American Mathematical Society* 76.1 (1954), pp. 92–111.

[109] D. M. Young and K. C. Jea. "Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods." In: *Linear Algebra and its applications* 34 (1980), pp. 159–194.

[110] J. W. Young. “A first order approximation to the optimum checkpoint interval.” In: *Comm. of the ACM* 17.9 (1974), pp. 530–531.

[111] M. Zounon. “Towards resilient parallel linear Krylov solvers: recover-restart strategies.” PhD thesis. INRIA, 2013.

List of publications

Articles in International Refereed Journals

[J1] J. Yin and Y. Du. "A class of preconditioners based on positive-definite operator splitting iteration methods for variable-coefficient space-fractional diffusion equations." In: *Communications on Applied Mathematics and Computation* (2021), pp. 157–176.

[J2] Y. Du, K. Hayami, N. Zheng, K. Morikuni, and J. Yin. "Kaczmarz-type inner-iteration preconditioned flexible GMRES methods for consistent linear systems." In: *SIAM Journal on Scientific Computing* (2021), S345–S366.

[J3] Y. Du, L. Marchal, G. Pallez, and Y. Robert. "Optimal checkpointing strategies for iterative applications." In: *IEEE Transactions on Parallel and Distributed Systems* (2021), pp. 507–522.

Articles in International Refereed Conferences

[C1] Y. Du, L. Marchal, G. Pallez, and Y. Robert. "Robustness of the Young/Daly formula for stochastic iterative applications." In: *49th International Conference on Parallel Processing-ICPP*. 2020, pp. 1–11.

[C2] 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, pp. 701–710.

Research Reports

[R1] Y. Du, L. Marchal, G. Pallez, and Y. Robert. *Optimal checkpointing strategies for iterative applications*. Research Report RR-9371. Inria - Research Centre Grenoble – Rhône-Alpes, 2020. URL: https://hal.inria.fr/hal-02980455.

[R2] Y. Du, L. Marchal, G. Pallez, and Y. Robert. *Robustness of the Young/Daly formula for stochastic iterative applications*. Research Report RR-9332. Inria - Research Centre Grenoble – Rhône-Alpes, 2020. URL: https://hal.inria.fr/hal-02514107.