HPC - Big Data Convergence : Managing theDiversity of Application Profiles on HPC Facilities

Valentin Honoré

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Valentin HONORÉ**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

HPC - Big Data Convergence: Managing the Diversity of Application Profiles on HPC Facilities

---

Date de soutenance : 15 Octobre 2020

Devant la commission d'examen composée de :

M. Gabriel ANTONIU	Directeur de recherche, Inria / Irisa	Président du jury
Mme Anne BENOIT	Maîtresse de conférences, ENS Lyon / LIP	Rapporteuse
Mme Ewa DEELMAN	Directrice de recherche, SC Information Sciences Institute	Examinatrice
M. Brice GOGLIN	Directeur de recherche, Inria / LaBRI	Directeur
M. Guillaume PALLEZ	Chargé de recherche, Inria / LaBRI	Encadrant
M. Frédéric SUTER	Directeur de recherche, CNRS / CC-IN2P3	Rapporteur

– 2020 –

Résumé Le calcul haute performance est un domaine scientifique dans lequel de très complexes et intensifs calculs sont réalisés sur des infrastructures de calcul à très large échelle appelées supercalculateurs. Leur puissance calculatoire phénoménale permet aux supercalculateurs de générer un flot de données gigantesque qu'il est aujourd'hui difficile d'appréhender, que ce soit d'un point de vue du stockage en mémoire que de l'extraction des résultats les plus importants pour les applications.

Nous assistons depuis quelques années à une convergence entre le calcul haute performance et des domaines tels que le BigData ou l'intelligence artificielle qui voient leurs besoins en termes de capacité de calcul exploser. Dans le cadre de cette convergence, une grande diversité d'applications doit être traitée par les ordonnanceurs des supercalculateurs, provenant d'utilisateurs de différents horizons pour qui il n'est pas toujours aisé de comprendre le fonctionnement de ces infrastructures pour le calcul distribué.

Dans cette thèse, nous exposons des solutions d'ordonnancement et de partitionnement de ressources pour résoudre ces problématiques. Pour ce faire, nous proposons une approche basée sur des modèles mathématiques qui permet d'obtenir des solutions avec de fortes garanties théoriques de leur performance. Dans ce manuscrit, nous nous focalisons sur deux catégories d'applications qui s'inscrivent en droite ligne avec la convergence entre le calcul haute performance et le BigData: les applications intensives en données et les applications à temps d'exécution stochastique.

Les applications intensives en données représentent les applications typiques du domaine du calcul haute performance. Dans cette thèse, nous proposons d'optimiser cette catégorie d'applications exécutées sur des supercalculateurs en exposant des méthodes automatiques de partitionnement de ressources ainsi que des algorithmes d'ordonnancement pour les différentes phases de ces applications. Pour ce faire, nous utilisons le paradigme in situ, devenu à ce jour une référence pour ces applications. De nombreux travaux se sont attachés à proposer des solutions logicielles pour mettre en pratique ce paradigme pour les applications. Néanmoins, peu de travaux ont étudié comment efficacement partager les ressources de calcul les différentes phases des applications afin d'optimiser leur temps d'exécution.

Les applications stochastiques constituent la deuxième catégorie d'applications que nous étudions dans cette thèse. Ces applications ont un profil différent de celles de la première partie de ce manuscrit. En effet, contrairement aux applications de simulation numérique, ces applications présentent de fortes variations de leur temps d'exécution en fonction des carastéristiques du jeu de données fourni en entrée. Cela est dû à leur structure interne composée d'une succession de fonctions, qui diffère des blocs de code massifs composant les applications intensive en données. L'incertitude autour de leur temps d'exécution est une contrainte très forte pour lancer ces applications sur les supercalculateurs. En effet, l'utilisateur doit réserver des ressources de calcul pour une durée qu'il ne connait pas. Dans cette thèse, nous proposons une approche novatrice pour aider les utilisateurs à déterminer une séquence de réservations optimale qui minimise l'espérance du coût total de toutes les réservations. Ces solutions sont par la suite étendues à un modèle d'applications avec points de sauvegarde à la fin de (certaines) réservations afin d'éviter de perdre le travail réalisé lors des réservations trop courtes. Enfin, nous proposons un profiling d'une application stochastique issue du domaine des neurosciences afin de mieux comprendre les propriétés de sa stochasticité. A travers cette étude, nous montrons qu'il est fondamental de bien connaître les caractéristiques des applications pour qui souhaite élaborer des stratégies efficaces du point de vue de l'utilisateur.

Mots-clés Calcul haute performance, applications stochastiques, applications à forte intensité en données, paradigme *in situ*, algorithmes d'ordonnancement, partitionnement de ressources, stratégies de réservations, *profiling* d'applications

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Title HPC - Big Data Convergence: Managing the Diversity of Application Profiles on HPC Facilities

Abstract Numerical simulations are complex programs that allow scientists to solve, simulate and model complex phenomena. High Performance Computing (HPC) is the domain in which these complex and heavy computations are performed on large-scale computers, also called supercomputers. For instance, cosmological simulations that aim to understand the physical behavior of dark energy and dark matter require simulating environments involving trillions of particles. To do so, these simulations require storage and computing capabilities that none of our personal computers are able to offer.

Supercomputers are complex computers designed to perform quadrillions of numerical operations per second for the most powerful of them. A supercomputer is hierarchical: it is composed of interconnected servers, containing interconnected processors. Each processor is composed of several cores, a limited amount of memory, as well as cables in order to interconnect all those components with the other servers. The central memory of the machine and the main storage capacities are hard-drives located in separated boxes. Such an internal organization implies a really complex and dense interconnection network.

Nowadays, most scientific fields need supercomputers to undertake their research. It is the case of cosmology, physics, biology ,or chemistry. Recently, we observed a convergence between Big Data/Machine Learning and HPC. Applications coming from these emerging fields (for example, using *Deep Learning* frameworks) are becoming highly compute-intensive. Hence, HPC facilities have emerged as an appropriate solution to run such applications.

From the large variety of existing applications has risen a necessity for all supercomputers: they must be generic and compatible with all kinds of applications. Indeed, it is hardly justified to design a machine that cost tens of millions of euros if only a few applications are able to run on it. Actually, computing nodes also have a wide range of variety, going from CPU to GPU with specific nodes designed to perform dedicated computations. Each category of node is designed to perform very fast operations of a given type (for example vector or matrix computation).

Supercomputers are used in a competitive environment. Indeed, multiple users simultaneously connect and request a set of computing resources to run their applications. This competition for resources is managed by the machine itself via a specific program called scheduler. This program reviews, assigns, and maps the different user requests. Each user asks for (that is, pay for the use of) access to the resources of the supercomputer in order to run his application. These resources are, typically, a subset of the available computational nodes with their associated features (memory and storage to store/load data, etc). The user is granted access to some resources for a limited amount of time. This means that the users need to estimate how many compute nodes they want to request and for how long, which is often difficult to decide.

In this thesis, we provide solutions and strategies to tackle these issues. We propose mathematical models, scheduling algorithms, and resource partitioning strategies in order to optimize high-throughput applications running on supercomputers. In this work, we focus on two types of applications in the context of the convergence HPC/Big Data: data-intensive and irregular (or stochastic) applications.

Data-intensive applications represent typical HPC frameworks. These applications are made up of two main components. The first one is called simulation, a very compute-intensive code that generates a tremendous amount of data by simulating a physical or biological phenomenon. The second component is called analytics, during which sub-routines post-process the simulation output to extract, generate, and save the final result of the application. We propose to optimize these applications running on supercomputers by designing automatic resource partitioning and scheduling strategies for both of its components. To do so, we use the well-known *in situ* paradigm that consists in scheduling both components together in order to reduce the huge cost of saving all simulation data on disks. While most of related works propose software solutions for *in situ* processing, we propose to include into these solutions automatic resource partitioning models and scheduling heuristics to improve the overall performance of *in situ* applications.

Stochastic applications are applications for which the execution time depends on its input, while in usual data-intensive applications the makespan of simulation and analytics are not affected by such parameters. Stochastic jobs originate from Big Data or Machine Learning workloads, whose performance is highly dependent on the characteristics of input data. These applications have recently appeared on HPC platforms. However, the uncertainty of their execution time remains a strong limitation when using supercomputers. Indeed, the user needs to estimate how long his job will have to be executed by the machine, and enters this estimation as his/her first reservation value. But if the job does not complete successfully within this first reservation, the user will have to resubmit the job, this time requiring a longer reservation. In the end, the total cost for the user will be the overall cost of all the reservations that were necessary to achieve the successful completion of the job. In this thesis, we propose to model the execution time of such applications by a probability distribution and to use this knowledge to derive an optimal reservation sequence. We also derive strategies including checkpointing at the end of some (well-chosen) reservations, to avoid wasting the benefits of failed reservations. Finally, we perform an in-depth profiling of such applications and show that a good knowledge of applications is critical when one aims to design cost-efficient strategies.

Keywords High Performance Computing, stochastic applications, data-intensive applications, *in situ* processing, scheduling algorithms, resource partitioning, reservation-based scheduling, application profiling

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Contents

List of Figures

List of Tables

Remerciements

Le temps est venu de conclure ces trois années de thèse par l'écriture de cette section de remerciements. Je présente par avance toutes mes excuses aux personnes que j'aurai pu oublier de mentionner, sachez que ce n'est en rien une volonté de ma part et que tous ceux que j'ai eu la chance de croiser au cours de mon parcours ont contribués à leur façon au succès de cette thèse.

Je commence par adresser toute ma gratitude aux membres du jury pour avoir accepté d'en faire partie en cette période sanitaire délicate. Merci à Anne et Frédéric pour la relecture attentive du manuscrit, qui a contribué grandement à l'amélioration de la version finale. Je ne peux imaginer la charge de travail que cela représente, surtout au cœur de l'été! Je remercie également Gabriel pour avoir accepté de présider le jury et d'avoir fait le déplacement jusque Bordeaux! Enfin, un grand merci à Ewa pour avoir accepté de se joindre à la soutenance tôt le matin! J'ai beaucoup apprécié les questions et remarques émises par chacun d'entre vous pendant la soutenance.

Les remerciements suivants seront pour Brice et Guillaume. Je vous suis infiniment reconnaissant pour la qualité de votre encadrement pendant ces trois années de thèse. Je m'estime très chanceux d'avoir pu travailler avec des chercheurs aussi talentueux. Brice, tes connaissances techniques m'impressionneront toujours, et ont grandement contribué à la qualité des travaux de ce manuscrit ainsi qu'à ma culture générale que j'ai acquise du domaine. Guillaume, je serai toujours impressionné par ton sens de l'intuition et tes grandes qualités de mathématiciens qui m'ont permis de travailler sur des thématiques passionnantes. J'aurai appris énormément de choses à vos côtés, et j'espère être aujourd'hui un meilleur chercheur que je ne l'étais. J'ai aussi beaucoup apprécié les moments partagés avec vous en dehors du cadre professionnel, même si ce fût à mon grand regret très limité sur la troisième année...

Un grand merci à tous les membres de l'équipe TADaaM. J'ai eu la chance de cotoyer des personnes bienveillantes venant d'horizon très différents. Merci à Emmanuel pour la direction de l'équipe (et les messages supprimés sur Mattermost :P), à Guillaume M. pour ses goûts cinématiques (parfois plus que douteux) et être la preuve vivante que la nourriture n'est pas le carburant de l'Homme, à François P. pour les discussions passionnantes, et à tous les autres permanent(e)s pour leur bienveillance quotidienne. Merci à tous les membres de l'open-space B427 pour la bonne humeur quotidienne qui y règne. Merci à Philippe et Andrès pour les stratégies (fouareuses ou noni à *The Crew*, à Valentin Hoy. pour avoir accepté de travailler sur *hwloc* pour que Brice ne m'y force

pas :), à Florian qui préfère la pollution parisienne à la douceur girondine, à Adrien pour jouer les 007 chez STORM, à Cyril B. pour son humour quotidien qui manque aujourd'hui à l'open-space (ou pas en fait!). Merci à Nicolas V. pour les bons passés ensemble (parfois de souffrance sportive héhé) pendant ces deux dernières années. Tu as largement ta brique dans l'édifice de cette thèse. Un merci général à tous ceux qui sont passés par le B427 pendant ces 3 dernières années, et félicitations à tous les docteurs!
Un énorme merci à Catherine pour son aide administrative ô combien précieuse!

Mon séjour à Inria fut des plus agréable et je tiens à remercier tous ceux que j'ai pu croiser au détour d'un couloir. Je ne remercie pas les joueurs de babyfoot (en particulier François R.) pour les humiliations nombreuses autour du baby :). Une pensée à tous les membres actuels ou passés du service communication/médiation du centre pour leur gentillesse. Je leur suis très reconnaissant de m'avoir permis de faire de la médiation (Fête du Centre, Fête de la Science, Déclics etc), chose à laquelle on prend vite goût et qui, je pense, est une activité très importante pour un chercheur. Un clin d'œil également à la médiathèque (Nathalie F. et tous les autres), véritable source de relaxation entre midi et deux.

Avant de conclure, je souhaite remercier tous les amis que j'ai rencontré depuis mes débuts à l'Université à Montpellier. On ne se voit pas assez souvent à mon goût, mais à chaque fois qu'on le peut c'est un plaisir indéniable. Merci à Ugo B. et Jimmy B. pour être resté si proche depuis ces débuts, le "gang" de l'ENS Lyon pour le soutien inconditionnel lors de la scolarité ENS puis le doctorat, Kshitij T. et Alessandro C. les amis rencontrés au Japon en M1 et qui me sont toujours aussi chers, Roberto G. et Bastin B. les collocs de première année de thèse (je n'oublierai jamais cet Italie-Suède de 2017i.
Un petit coucou à la section Kung-Fu du dojo Bordeaux, merci à Benoît et Juliette de ne pas désespérer de mon niveau technique et de ma souplesse :).

Je souhaite terminer cette section remerciement par ma famille qui m'a toujours soutenue, que ce soit dans mon changement d'orientation au lycée ou dans mon idée de rejoindre l'ENS Lyon. Un immense merci à mes parents qui ont toujours été là pour moi, et qui m'ont toujours encouragé, soutenu et protégé tout au long de ces années. Je leur serai à jamais reconnaissant. Un grand merci à ma sœur Sarah pour ses encouragements nombreux tout au long de cette thèse, et pour avoir accepté la (très) lourde tâche de relire une majeure partie de ce manuscrit, ce qui en a très grandement amélioré la qualité littéraire! Un énorme merci :)
Une grande pensée pour mes grands-parents qui ont toujours été là pour moi, pour m'encourager et me soutenir. À toute ma famille, je suis très fier de faire partie de la vôtre!
Une pensée pour Milo, Vénus, et Eliot qui je l'espère, au paradis des chiens, profitent de ballades infinies. Votre perte aura été la plus douloureuse épreuve à surmonter.
Enfin, merci Quyen pour être à mes côtés depuis maintenant quasiment 5ans. Merci pour tous les sacrifices consentis pendant cette thèse, à cause de mes sessions de travail beaucoup trop tardives ou des déplacements/visites à l'étranger. Je me sens vraiment redevable d'avoir eu une personne aussi compréhensive et attentive à mes côtés pendant cette thèse. Val cảm ơn nha!

Résumé étendu en français

Le calcul haute performance est un domaine scientifique dans lequel de très complexes et intensifs calculs sont réalisés sur des infrastructures de calcul à très large échelle appelées supercalculateurs. De nombreux domaines scientifiques ont de nos jours recours aux simulations numériques afin de modéliser, simuler et expliquer les phénomènes naturels parmi les plus complexes de notre monde. Les programmes exécutés par les supercalculateurs sont des applications scientifiques très sophistiquées qui requièrent une puissance de calcul phénoménale afin de résoudre un problème d'intérêt sous-jacent. Ces applications scientifiques ne peuvent être exécutées sur nos ordinateurs personnels de par leurs énormes besoins à la fois en termes de ressources de calcul mais aussi de consommation mémoire. Par exemple, certaines simulations de cosmologie visent à étudier les propriétés physiques de la matière noire, qui serait une des clés pour mieux appréhender le fonctionnement de notre univers. Ces simulations nécessitent de reproduire des environnements dans lequels des trillions de particules interagissent entre elles.

Les supercalculateurs sont aujourd'hui devenus un outil central pour de très nombreux domaines scientifiques tels que la physique, la chimie, la biologie, les sciences de la terre ou la cosmologie. De ce fait, une très grande variété d'applications sont exécutées sur les supercalculateurs, chacune ayant ses propres propriétés et contraintes. À ce large spectre d'applications se sont récemment ajoutés de nouveaux programmes provenant de domaines connexes au calcul haute performance tels que le *BigData*, qui propose des solutions pour le traitement de données massives, ainsi que l'intelligence artificielle. En effet, les besoins en calcul de ces domaines sont devenus si élevés que seules les infrastructures à grand échelle proposées par le calcul haute performance peuvent satisfaire ces demandes. En particulier, l'apprentissage profond requiert d'entraîner et traiter d'énormes quantités de données afin d'instancier les modèles prédictifs des réseaux de neurones. Ainsi, nous assistons depuis quelques années à une convergence entre le calcul haute performance et les domaines du BigData et de l'intelligence artificielle.

Cette très grande diversité d'applications soulève une importante nécessité pour les supercalculateurs: ils doivent être génériques et compatibles avec le plus grand spectre possible d'applications. En effet, il est difficilement concevable de construire une infrastructure de calcul pour un coût de plusieurs dizaines de millions d'euros si seulement un petit nombre d'applications peut en bénéficier. Afin d'offrir cette flexibilité, les supercalculateurs se sont équipés d'un large éventail de nœuds de calcul, allant du processeur aux cartes graphiques en passant par les accélérateurs. Chaque type de ressource de calcul est optimisé pour certaines opérations calculatoires. Par exemple, les processeurs graphiques sont reconnus pour leurs performances dans le calcul matriciel.

Ainsi, les supercalculateurs sont des machines de calcul ayant une architecture très complexe. Ces machines sont organisées en boîtes hiérarchiques interconnectées. Chaque boîte contient des unités de calcul. Les processeurs, unité de calcul la plus répandue, contiennent différents cœurs de calcul ainsi qu'une quantité restreinte de mémoire volatile utilisée pour stocker de manière temporaire les données de calcul. Un supercalculateur contient aussi une mémoire centrale persistente, également centralisée dans une boîte, et composée de disques durs sur lesquels peuvent être sauvegardées des données de manière pérenne. Enfin, un réseau de communication performant assure la connexion entre les différents éléments d'une même boîte ainsi qu'un accès vers les autres composants du supercalculateur. Une telle organisation génère un réseau de communication très dense et complexe. De par le nombre de processeurs présents dans les supercalculateurs, ceux-ci sont capables d'effectuer jusqu'à 10^{15} opérations arithmétiques par seconde pour les plus puissants d'entre eux au moment de l'écriture de cette thèse. Cette puissance calculatoire phénoménale permet aux supercalculateurs de générer un flot de données gigantesque qu'il est aujourd'hui difficile d'appréhender, que ce soit d'un point de vue du stockage en mémoire que de l'extraction des résultats les plus importants pour les applications.

Au delà de leur architecture complexe, les supercalculateurs sont aussi un environnement très compétitif lorsqu'il s'agit d'accéder à leurs ressources. En effet, de nombreux utilisateurs peuvent simultanément se connecter à la machine et réserver un ensemble de ressources de calcul pour exécuter leurs applications. Afin d'avoir un accès à la machine, les utilisateurs doivent auparavant effectuer une procédure de demande d'accès aux administrateurs du supercalculateur en motivant leur besoin en calcul. Souvent, cet accès est garanti pour la soumission d'un projet scientifique démontrant le besoin de la machine pour son accomplissement. La compétition pour les ressources de la machine est gérée au niveau de la machine elle-même, par un programme complexe et sophistiqué appelé ordonnanceur. Ce programme est en charge de recevoir les différentes requêtes des utilisateurs, de les arbitrer et de faire en sorte de couvrir simultanément le plus possible de requêtes utilisateurs avec les ressources physiques de la machine. Chaque utilisateur envoie une demande en nombre de ressources de calcul à l'ordonnanceur (concrètement, l'utilisateur paye pour l'utilisation de ces ressources) pour une durée déterminée, afin d'exécuter son application. Typiquement, l'utilisateur demande l'accès à un sous-ensemble des ressources de calcul disponibles sur la machine (processeurs, processeurs graphiques, accélérateurs) avec leurs fonctionnalités associées (accès à leur mémoire locale et aux disques durs). L'utilisateur peut alors se voir affecter ces ressources pour un temps prédéfini, indiqué à l'ordonnanceur lors de la requête initiale. Cela signifie que les utilisateurs doivent être en mesure d'estimer assez précisément la quantité de nœuds de calcul dont ils ont besoin, mais aussi pour combien de temps ces ressources leur seront affectées. Avec ces différentes requêtes, l'ordonnanceur construit alors un ordonnancement des différentes applications, et se charge de distribuer les ressources aux utilisateurs en fonction de cet ordonnancement. Ce processus est itéré de manière dynamique, avec une réévaluation de l'ordonnancement à chaque nouvel évènement (complétion d'une réservation, nouvelle requête etc). Les requêtes en attente de traitement sont placées dans une ou plusieurs files d'attente de l'ordonnanceur, chacune ayant une évaluation propre de la priorité à donner aux requêtes. Ainsi, l'objectif de l'ordonnanceur est évidemment de maximiser l'utilisation de toutes les ressources de calcul afin d'optimiser le rendement de la machine tout en guarantissant une équité maximale entre les utilisateurs.

Dans le cadre de la convergence entre le calcul haute performance et les domaines avec des besoins en calcul grandissant, une grande diversité d'applications doit être traitée par les ordonnanceurs des supercalculateurs, provenant d'utilisateurs de différents horizons pour qui il n'est pas toujours aisé de comprendre le fonctionnement de ces infrastructures pour le calcul distribué. Or, l'estimation du besoin en ressources de calcul des applications est fondamental pour la performance à la fois des applications des utilisateurs mais aussi des machines elles-mêmes. Prenons l'exemple d'un utilisateur qui surestime le temps de réservation nécessaire pour l'exécution de son application. Cette dernière se termine donc avant la durée totale de réservation des ressources. Dans ce cas, ces ressources sont libérées au profit d'autres utilisateurs. De ce fait, l'ordonnanceur de la machine se retrouve avec un trou au milieu de son ordonnancement, qu'il va essayer de combler en sélectionnant des requêtes d'utilisateurs dans ses files d'attente. Ce processus, appelé *backfilling*, ne suffit souvent pas à combler les différents vides laissés par les surestimations du temps d'exécution des applications. Quant à l'utilisateur, le coût associé à sa réservation est plus élevé que ce qu'il aurait pu payer avec une réservation plus proche de la réalité des besoins de son application. Dans le cas contraire d'une sous-estimation (temps d'exécution de l'application plus long que la durée de réservation des ressources), les conséquences pour l'utilisateur sont importantes car son application est brutalement interrompue à la fin de la réservation. Cet arrêt soudain de l'application peut engendrer la perte du progrès réalisé durant la réservation si l'utilisateur n'a pas prévu de sauvegarde de ses données au cours de l'exécution. De fait, cette réservation est perdue à la fois pour l'utilisateur qui aura payé une réservation qui ne lui aura pas été bénéfique, et pour la machine qui n'a pas utilisé une partie de ses ressources pour effectuer des calculs utiles. Les exemples précédents démontrent à quel point l'estimation de la durée des réservations est importante pour optimiser le fonctionnement des supercalculateurs, ainsi que limiter le coût des réservations pour les utilisateurs. Ainsi, ces utilisateurs doivent réaliser des réservations les plus représentatives possibles des besoins réels de leurs applications, qui peuvent être difficiles à estimer en pratique.

Dans cette thèse, nous exposons des solutions d'ordonnancement et de partitionnement de ressources pour résoudre ces problématiques. Pour ce faire, nous proposons une approche basée sur des modèles mathématiques qui permet d'obtenir des solutions avec de fortes garanties théoriques de leur performance.

Dans ce manuscrit, nous nous focalisons sur deux catégories d'applications qui s'inscrivent en droite ligne avec la convergence entre le calcul haute performance et le *BigData*: les applications intensives en données et les applications à temps d'exécution stochastique.

Les applications intensives en données représentent les applications typiques du domaine du calcul haute performance. Ces applications sont classiquement décomposées en deux parties. La première est appelée simulation, et se présente comme un code calculatoirement très intensif qui s'exécute en parallèle sur de nombreuses ressources de calcul. Ce code est développé afin d'étudier et reproduire des phénomènes issus du monde réel provenant des domaines scientifiques mentionnés au début de ce résumé. De par le nombre important de ressources de calcul sur lesquelles elle s'exécute, la simulation est capable de génèrer d'énormes volumes de données. La deuxième partie de l'application est appelée analyse. Pendant cette phase, des sous-routines réalisent un post-traitement des données de la simulation. Ainsi, l'analyse des données permet de réaliser des opérations d'extraction, de génération de statistiques ou de sauvegarde des données. L'analyse est une partie très importante de l'application qui permet à la fois de vérifier l'intégrité des données mais aussi de générer le résultat final de l'application. De manière générale, l'analyse est également employée pour effectuer des opérations de visualisation du problème sous-jacent tout au long de la simulation afin de permettre à l'utilisateur de bénéficier d'un affichage interactif du progrès de l'application, voire d'influer sur son exécution. Dans cette thèse, nous proposons d'optimiser cette catégorie d'applications exécutées sur des supercalculateurs en exposant des méthodes automatiques de partitionnement de ressources ainsi que des algorithmes d'ordonnancement pour les différentes phases de ces applications. Pour ce faire, nous utilisons le paradigme *in situ*, devenu à ce jour une référence pour ces applications. Le principe de ce paradigme est de coupler l'analyse à la simulation afin de post-traiter les données directement sur les nœuds de simulation où elles sont produites. Cela évite le stockage en mémoire de l'ensemble des données de simulation pour les traiter une fois la simulation terminée. Avec les énormes volumes de données générés par la simulation, cette étape de stockage est aujourd'hui rédhibitoire du fait des performances très variables des opérations d'accès aux disques, ainsi que de la taille exponentielle des volumes de données à stocker. De nombreux travaux se sont attachés à proposer des solutions logicielles pour mettre en pratique ce paradigme pour les applications. Néanmoins, peu de travaux ont étudié comment partager efficacement les ressources de calcul entre la simulation et l'analyse afin d'optimiser le temps d'exécution des applications. Nous proposons dans la première partie de cette thèse des stratégies de partitionnement de ressources associées à des algorithmes d'ordonnancement des différentes analyses afin de les coupler avec la simulation.

Les applications stochastiques constituent la deuxième catégorie d'applications que nous étudions dans cette thèse. Ces applications ont un profil différent de celles de la première partie de ce manuscrit. En effet, contrairement aux applications de simulation numérique, ces applications présentent de fortes variations de leur temps d'exécution en fonction des carastéristiques du jeu de données fourni en entrée. Cela est dû à leur structure interne composée d'une succession de fonctions, qui diffère des blocs de code massifs composant les applications intensive en données. Les applications stochastiques émergent de domaines tels que l'intelligence artificielle ou le *Big Data*. L'incertitude autour de leur temps d'exécution est une contrainte très forte pour lancer ces applications sur les supercalculateurs, comme mentionné ci-avant dans ce résumé. Afin d'exécuter ces applications, un utilisateur doit réaliser une première réservation dont il doit estimer la durée. Si l'application n'a pas pu se terminer pendant cette réservation, alors cet utilisateur devra réaliser une seconde réservation d'une durée supérieure à la première. Ce processus devra être répété jusqu'au succès de l'application. Au final, le coût pour l'utilisateur sera la somme des coûts de chaque réservation ainsi réalisée. Dans cette thèse, nous proposons une approche novatrice pour aider les utilisateurs de ce type d'application à calculer une séquence de réservations optimale qui minimise l'espérance du coût total de toutes les réservations. Pour ce faire, nous proposons une modélisation mathématique rigoureuse de ces applications en représentant leur temps d'exécution par une distribution de probabilité. Par la suite, nous utilisons les propriétés mathématiques de cette distribution afin de calculer une séquence optimale de réservations. Ces solutions sont par la suite étendues à un modèle d'applications avec points de sauvegarde à la fin de (certaines) réservations afin d'éviter de perdre le travail réalisé lors des réservations trop courtes. Enfin, nous proposons un *profiling* d'une application stochastique issue du domaine des neurosciences afin de mieux comprendre les propriétés de sa stochasticité. À travers cette étude, nous montrons qu'il est fondamental de bien connaître les caractéristiques des applications pour qui souhaite élaborer des stratégies efficaces du point de vue de l'utilisateur.

Introduction

High Performance Computing (HPC) refers to the performance of complex and heavy computations on large-scale computers called supercomputers. The programs that are executed on such machines are called applications. Each application aims at solving, simulating, or modeling complex phenomena that a simple computer would not be able to handle due to the complexity and size of the target problem. For instance, cosmological simulations aiming at understanding the physical behavior of dark energy and dark matter requires simulating environments involving trillions of particles [69]. To do so, these simulations require storage and computing capacities that no personal computers could possibly offer. Simulations are really complex programs, often composed of sub-tasks that are processed in a very specific order with strong dependencies on one another. An application can be compared to a cooking recipe: the cake represents the target result, while the different steps of the recipe corresponds to the different sub-routines of the application that need to be performed in order to get the final result.

Nowadays, most scientific fields need supercomputers to undertake their research. It is the case of cosmology, physics, biology or chemistry. From the large variety of existing applications has risen a necessity for all supercomputers: they must be generic and compatible with all kinds of applications. Indeed, it is hardly justified to design a machine that cost tens of millions of euros[1] if only a few applications are able to run on it. To offer this universality, many software runtimes and libraries are designed to offer a wide range of flexibility in terms of applications. This set of software is called a software stack because it offers services at many different levels of the program execution (for instance within the application itself or to ease interactions between programs and machine).

From megascale to exascale computers...

Most countries or companies involved in HPC research have developed more and more powerful supercomputers through years, following the breakthrough in hardware design and the new computational needs due to the progress of scientific research and the industrial evolution. The TOP500 [4] is a statistical ranking showing the characteristics and performance of the 500 most powerful machines in the world. The performance of such machines are evaluated thanks to the LINPACK benchmark[2], which resolves a system of linear equations. Machine performance are assessed by determining how many FLoating-Point Operations Per Seconds (FLOPS) it is able to perform. As of June 2020, the most powerful supercomputers are nowadays able to reach around 400 petaFLOPS (400×10^{15} FLOPS) in terms of computation power, which equates to assembling tens of millions of laptops together.

Over the past years, supercomputers have become extremely powerful in terms of computational power. Let us provide a brief historical background and summary of

[1] The cost of Aurora, the first exascale machine to be developed at Argonne National Laboratory is estimated to more than $500M [5].

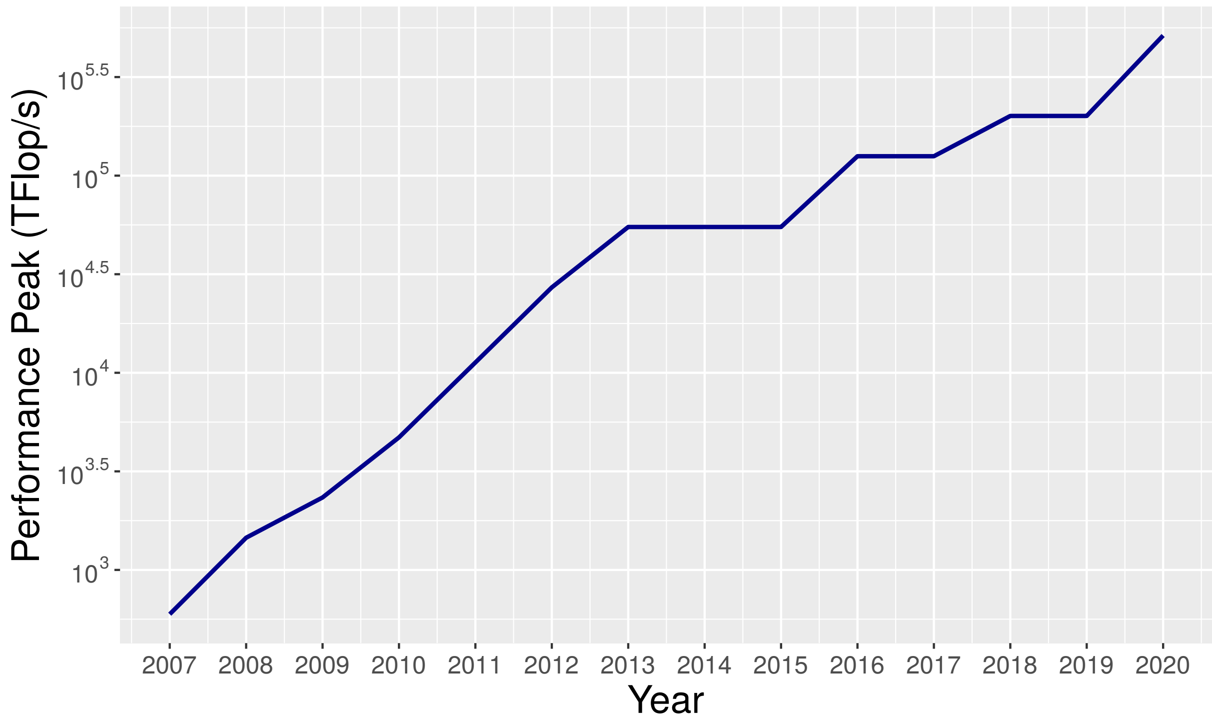
[2] https://www.top500.org/project/linpack/

Figure 1: Supercomputer Cray-1 (photo credits from `https://www.cray.com`)

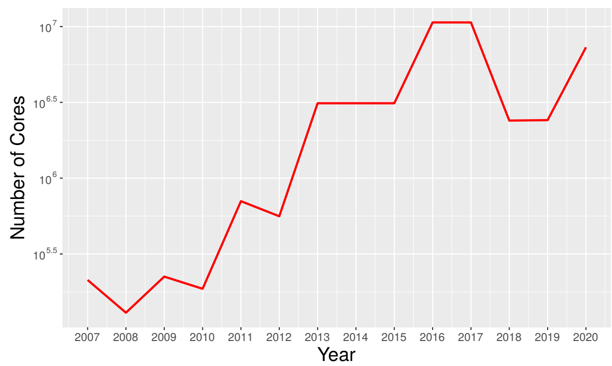
evolution of supercomputers. The first supercomputer was the Cray-1, illustrated in Figure 1. It was designed by Cray company in the 70s. Up to now, it remains the most successful supercomputer of history, with more than 80 machines sold. It is the first computer to be made up of several boxes and linked together with wires arranged into a ring. The idea of separating the computer into modules came from the difficulty of adding more and more components to a single module, making the architecture too complex. The Cray-1 computer was able to reach 160 megaFLOPS (that is, 160 millions operations per second) at peak performance. During the 90s, hardware enhancements allowed supercomputers to be built with thousands of processors organized into clusters. In 1997, the supercomputer Intel ASCI Red/9152 from DoE Sandia National Laboratory (USA) was the first computer ranked in the TOP500 with teraFLOPS computing power. From then on, the era of modern supercomputers starts and the power of supercomputers keeps increasing to reach petascale, and soon exascale with machines such as Frontier [7] or Aurora [6] in 2021 (which is expected to propose more than 1.5 exaFLOPS, or 1.5 trillion FLOPS).

In 2012, for the first time in history, a supercomputer (the Sequoia machine at the Lawrence Livermore National Laboratory, USA) reached a million cores according to TOP500 ranking. The cores are the most basic compute units of supercomputers. Hence, the quantity of cores is a good metric to evaluate the degree of parallelism associated to a machine. As of August 2013, the top five supercomputers all feature more than 500,000 cores.

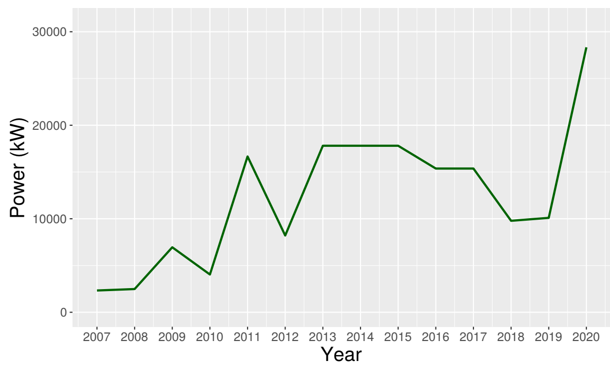
Figure 2 shows the evolution of machine peak performance (Figure 2a), the number of cores (Figure 2b) and the total power (Figure 2c) of the machine ranked first at Top500 from 2007 to June 2020, date of the last update of the ranking at the time this manuscript is released. We can see that, in this period, the performance peak has been multiplied by 330, while the machines have multiplied their number of cores by a factor 10. These



(a) Machine Peak Performance (TFlop/s)



(b) Number of Cores



(c) Total Power (kW)

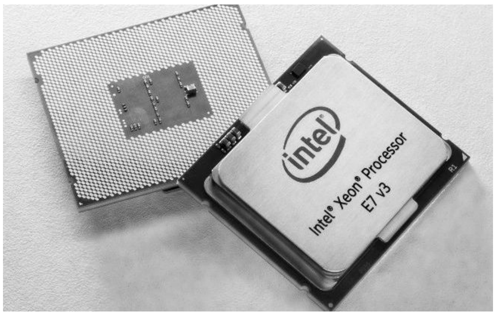
Figure 2: Evolution of the machine ranked 1st in TOP500 for some selected features. Data are extracted from November 2007 to June 2020.

figures show all the progress that has been done in terms of hardware performance. Following this trend, the energy needs of these machines have become quite significant. This is not only due to the multiplication of hardware components, but also to the necessity of cooling the whole system. Altogether, these growing energy requirements lead to a higher energy consumption. For instance, the Frontier exascale supercomputer is expected to have a power consumption in the range of 30 MW [7], which represents the energy consumption of 24,000 households according to the U.S. Energy Information Administration [10]. In addition to the energy cost, maintenance is also a major criteria since it is necessary to be able to maintain and repair hardware components. Several other costs have to be envisioned too, such as staffing or software licences. Altogether, supercomputers represent at the same time a major tool for scientists to solve their problems and an important cost for the structure that hosts them.

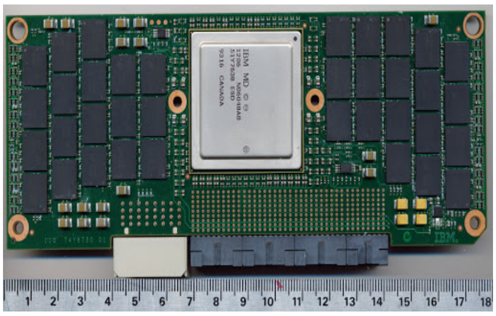
Just like our personal computers, supercomputers all rely on the Von Neumann architecture [145]. In essence, this model describes the programs running on a computer as a sequence of instructions that are processed by a process unit and managed by a control unit. The programs are loaded into the memory of the machine, which can also be accessed by the programs to store data. However, supercomputer's design is more specific because it should be able to manage millions of process units at the same time. To do so, supercomputers have a hierarchical layout.

Indeed, a supercomputer is composed of boxes nested in each other. Figure 3 illustrates the architecture of a supercomputer. The basic box is the process unit (Figure 3a). It is the component that allows the machine to perform basic arithmetic and controlling operations. Different types of units can be found depending on the computations that they have been optimized to perform. Among classical examples are the Central Processing Unit (CPU), the Graphical Processing Unit (GPU), or accelerators. Several of these units (of the same type) are assembled together on a compute card (Figure 3b). Then, compute cards are interconnected into a node board, and all node boards are then assembled together into a rack (Figure 3c). Then, racks are linked together into a cabinet (Figure 3d). Finally, several cabinets are connected to each other to form a supercomputer. Figure 3e shows the supercomputer *Mare Nostrum 2* of the Barcelona Supercomputing Center.

To put in a nutshell, a supercomputer is a set of hierarchical boxes, all interconnected with each others at different levels of the hierarchy. According to the Von Neumann infrastructure, memory is also a fundamental feature of supercomputers. The central persistent memory of the machine, called Parallel File System (PFS), is the biggest volume where each user has a dedicated storage space. Each memory volume is accessed by a communication bus that processes the writing and reading of memory instructions, called Input/Output operation (I/O) operations. Each communication bus has a limited traffic capacity that we call bandwidth. Alongside the PFS, memory spaces are available for users at each level of the machine hierarchy. These units can be accessed by the user programs running on the machine. Recently, some near-node persistent storage are also available in supercomputer infrastructures [7] to offer the users the possibility to manage their data with flexibility. For instance, data with high frequency access can



(a) Process Unit



(b) Compute Card

(c) Node Board

(d) Cabinet (or rack)

(e) Supercomputer

Figure 3: Structure of supercomputers: a set of interconnected hierarchical boxes.

be persistently stored in disks that are close to compute nodes in order to guarantee fast I/O operations.

In brief, supercomputers are infrastructures with a complex hardware layout. They require huge pecuniary investments in order to build them, and then maintain them over their lifetime.

... that applications need to accommodate

Usually, scientific applications are based on models of real-life phenomena (for instance, physics models). Because it is difficult to study these problems in laboratory conditions (either due to the impossibility to create a proper environment or its huge financial cost), models of these phenomena tend to be implemented into sophisticated computer programs that are designed to reach an important degree of parallelism in their codes in order to increase computational performance. This specific code structure makes large-scale machines the natural candidates to host such programs.

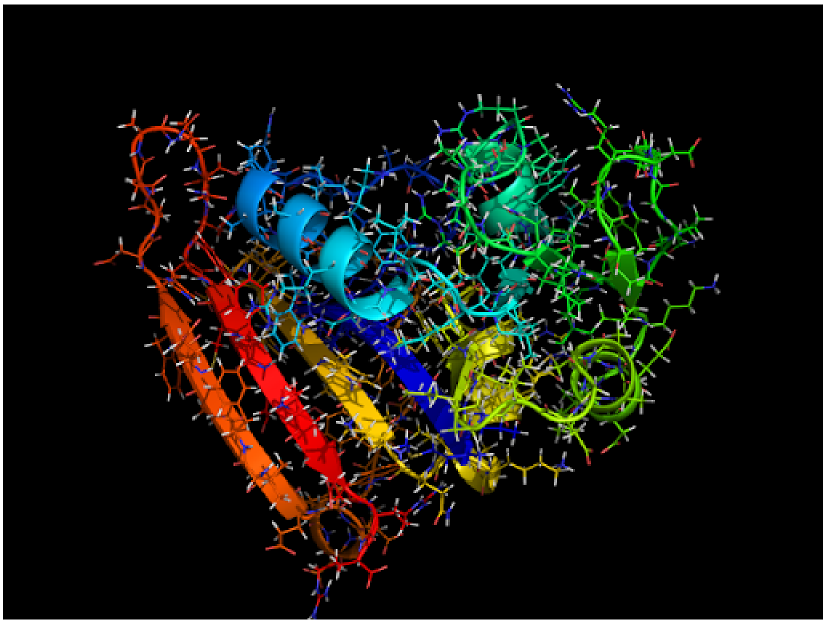


Figure 4: Illustration of molecular dynamics for proteins using Gromacs [3] software.

We already mentioned cosmological simulations that simulate environments with trillions of particles. Other domains, such as molecular dynamics, also use supercomputers to simulate proteins folding and unfolding. Figure 4 presents an example of a protein folding simulation using Gromacs package [3]. This type of computation requires several processes running in parallel with synchronization in order to generate this output. To do so, application developers make the use of most of the functionalities offered by the software stack. Some of these solutions are now widely used in most of HPC facilities. For instance, the Message Passing Interface (MPI) is the *de facto* standard that defines routines to perform communications in between processes of the same program running in parallel on allocated cores and nodes.

With modern architectures, scientific workflows generate a tremendous amount of data. Indeed, simulation codes have become rather sophisticated and tend to generate huge quantities of data, which are analyzed in a second phase in order to extract the desired results. For instance, a run of Hardware/Hybrid Accelerated Cosmology Code (HACC) [69], a cosmological toolkit, generates PetaBytes (PB) of data when Summit, ranked first at Top500 in 2019, has 250 PB of file system capacity. Such amount of data is complex to store on the PFS, not only because of its total volume, but also because of the bottleneck that represents the limited bandwidth of the communication bus. Between November 2010 and November 2020, the machine ranked first of the Top500 ranking saw its central storage capacity increased by a factor 25[3], while the interconnect speed only increased by less than a factor 2[4]. Hence, the growth factor of the speed of the interconnect is much smaller than the one of the storage capacity. While more and more storage capacity tends to be proposed, the performance ratio between the total capacity of the disks and the speed of access to it rather tends to deteriorate.

Basically, all data generated by an application are stored on the disks to be post-processed in a second phase. However, the limited traffic speed to access the disks had become a major restriction to the performance of the applications. Facing this pressure over the access to data storage, application users and developers have been forced to rethink the data life cycle. Therefore, data management has become a major concern for them over the last years. Users must extract or transform data so that the final volume to be saved on disk is as compact as possible. This fact has forced the HPC community to widen its perspective and explore solutions coming from other scientific fields.

... coming from different domains

Simultaneously, other domains face a dramatic increase of their computational needs. Artificial Intelligence (AI) and BigData are one of the best examples of this rise. For instance, Machine Learning, in particular Deep Learning, has an important computational power need. Indeed, important efforts of parallelization have been made in the training phase so that the networks are trained over bigger and bigger datasets. As a reason, Deep Learning users start to target HPC infrastructures to execute their applications. Moreover, those improvements have convinced some HPC users to include AI techniques in their applications. This integration is done at different levels of application, going from data pre-fetching to enhancement of computational load distribution and for else output analysis. BigData field brings expertise in data management and

[3]Jaguar [8] supercomputer proposed a 10PB size for the PFS, while Summit [9] has 250PB capacity.

[4]Single link capacity for Jaguar [8] had a maximum peak performance of 57.6 GBytes per second while the specifications of Summit [9] indicate 100 GBytes per second

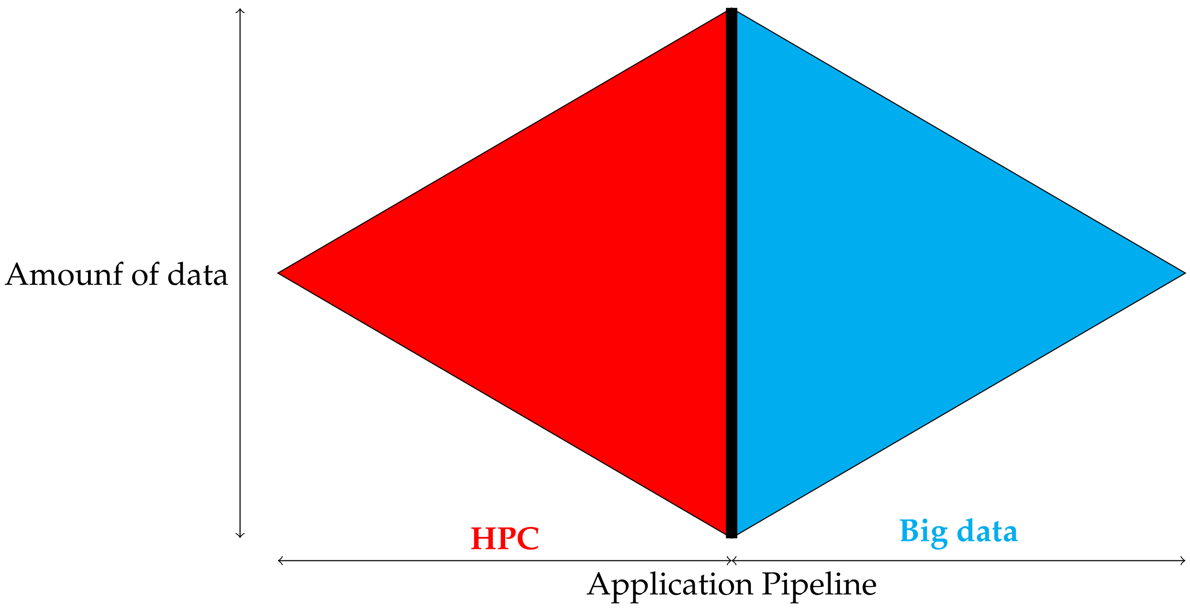


Figure 5: Convergence between HPC and BigData

visualization for HPC applications. Recently, Big Data applications are using compute-intensive procedures to perform data mining. Just like HPC application, this rise of computational requirements is due to the usage of modern AI techniques.

Consequently, we observe a convergence between these different domains with each are expecting to benefit from the others' innovations to solve its own limitations. Figure 5 illustrates this convergence since in this example HPC is used to generate an important amount of data, that the BigData field is able to treat in a second phase. Subsequently, HPC benefits from the BigData's expertise in data management, while BigData can use HPC facilities to improve its solutions.

Altogether, from an HPC perspective, such a convergence implies that supercomputers become the target for applications coming from many different fields, each having their own communities and development processes. For instance, data analysis (called analytics) is expected to become a full component of the HPC software stack. These analytics procedures directly come from BigData frameworks that process massive amounts of data. Thus, HPC infrastructures have to manage all these applications possibly running at the same time, on different types of compute units with very different needs. Some of these emerging applications feature a profile which is quite different from the one of the classical HPC applications. This implies that HPC infrastructures should not only be flexible in terms of hardware solutions, but also in terms of software offering.

Applications are executed on supercomputers by users who have been granted access to the machine. This access is usually provided after submission of a research project that demands a computational support. Candidate projects indicate a desired budget of computations (in cores $\times$ hours). Typically, if one has a total budget of 100 cores $\times$ hours, it means that it is possible to compute 100 hours on a single core. If with the same budget one books two cores of the machine, the total computation duration will be 50 hours. As a matter of fact, this time budget represents the financial cost associated to the total core per hours granted on the machine. In the end, the scientific board of the infrastructure reviews the submissions and either accepts or rejects user's access requests. We depict in Figure 6 a schematic representation of the organization of a large-scale infrastructure.

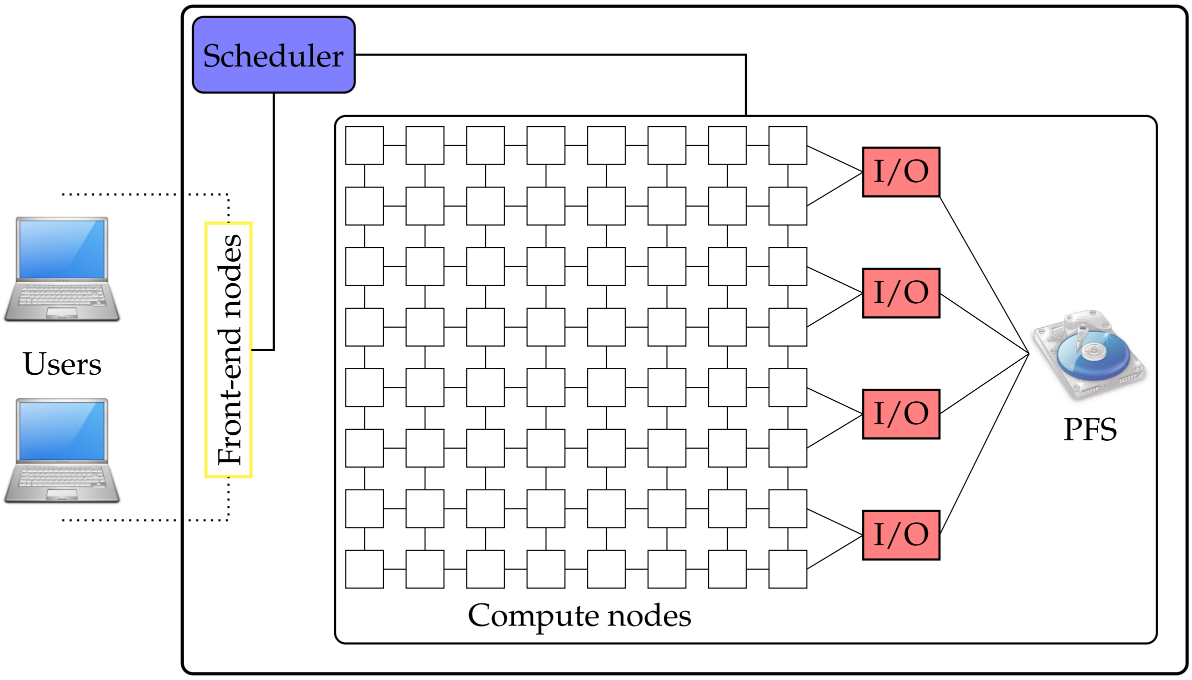


Figure 6: Representation of a typical large-scale infrastructure. Users access the machine by the front-end nodes and perform submissions to the scheduler. Access to PFS is one of the major bottleneck for performance due to limited capacities of the storage network compared to the dimension of the resources available for computations.

Once the access is granted, users can run their programs by booking a subset of the computational resources of the machine for a defined duration. However, hundreds of users may want to start a program at the same time. As the resources of the machine are finite, it is not possible to allow a set of requests with a total number of resources greater than the machine's capacity. Hence, it is necessary to referee all the different demands. This arbitration process is performed by a very complex program called scheduler. At any time, the scheduler maps the requests (or jobs) of users basing on the resource availability of the system.

The scheduler's performance is fundamental in supercomputer operations. It guarantees fairness between users: a user should not wait infinitely before his request is actually allocated on the machine. In addition to fairness, scheduler must maximize system utilization at any time. Indeed, any compute unit that is not being dedicated to a request is costly for the infrastructure since it still runs at minima (waiting for instructions to process) without any user covering the expenses.

In the same fashion as there is a competition for compute units, users get caught up in a critical struggle for memory access. Indeed, we previously mentioned that the communication bus has a limited traffic bandwidth. When several applications are running at the same time on the machine, many I/O operations to access the PFS have to be processed in parallel of each others, which creates contention due to the finite nature of the bandwidth. Therefore, I/O bandwidth needs to be shared between several users. In Figure 6, we clearly see a difference of dimension between the number of resources available to perform computations and the size of the network to access the PFS. Even though some nodes are dedicated to perform I/O operations, their number is in fact insufficient in comparison with the needs of the different machine users, hence generating a bottleneck in the system. Moreover, I/O operations are often blocking for programs. It implies that, until the operations succeed, the application is waiting and does not perform any more computations. Consequently, I/O operations can slow down applications and directly impact their performance.

Altogether, supercomputers form a very competitive environment for users. Users send requests to access the resources of the machine, requests that are managed by the scheduler. Possibly, jobs can wait in the queue of the scheduler before being actually processed on the machine. From a user perspective, the objective is to see one's requests running in the machine as early as possible, which entails minimizing the time spent in the scheduler queue. From a system administrator perspective, the objective is to maximize the resource utilization of the machine. If all the compute units are constantly allocated to particular jobs, then the machine is fully profitable. Given the above, users and system administrators have different goals and expectations with regards to a supercomputer infrastructure.

Issue in question

In this introduction, we presented the main features of modern HPC facilities. Supercomputers offer a huge computational capacity at the cost of a complex architecture. Also, users are running their programs concurrently on the machine. This leads to a competition for resources (for instance, disk bandwidth) that can significantly degrade application performance.

On the one hand, there are users with a wide range of applications developed by different communities (AI, physics, etc). On the other hand, we have supercomputers that offer solutions to execute those applications. These solutions are twofold: high flexibility in hardware components (CPU, GPU, accelerators) and specific tools provided by software stack. However, due to their quite complex architectures, application developers and users must be able determine the needs of the applications in terms of compute nodes and memory. These decisions can either be performed statically before starting the application or evaluated dynamically during its execution. The mapping of application needs onto machine features is often difficult because of the complexity of the applications, which function by following several steps that are either parallel or sequential. For example, sequential applications require only one compute core, while parallel applications can benefit from several resources. In the case of sequential applications, assigning multiple nodes to the performance of this application is a waste of resources, since such application will only be using a single core. As a result, users must have an idea of the profile of their applications, that is to say of the different needs in terms of resources that will be required by the programs during their execution.

Therefore, the central question of this thesis is as follows: how to optimize various application profiles on HPC infrastructures? A natural answer to this question is to accurately estimate the needs of the different applications. With a good knowledge of one's application, one can efficiently map computational needs and machine features. This theory forms the core of the research carried out in this thesis.

Our approach consists in using mathematical models of the applications running on HPC facilities. By doing so, we aim at estimating application needs, both in terms of computational resources and memory. In addition to application models, we propose flexible platform abstractions through expressive models of HPC facilities.

Since the applications running on supercomputers belong to many different scientific domains, their profile can be very different. For this reason, creating a generic model for all applications is a difficult task. To solve this issue, we propose to categorize applications by batching those that share common features. Then, for each category, we propose dedicated solutions in order to optimize each type of applications for HPC infrastructures.

Modeling application behavior requires a good knowledge of its workflow. Relevant informations regarding applications and their functioning can be directly provided by program developers or even by users through their past experience in the use of the programs in question. Another possibility is to directly study the application's behavior by performing tests and runs in order to build a profile of its features.

The advantage of our approach is that our solutions are designed to be applicable to all applications of a same category. Likewise, our solutions are valid for many different uses of the applications, without consideration of the nature of the input or of the scientific problem to be solved. Indeed, scientific codes often present many parameters that are used during the computation of the problem under study. By providing a generic model of the applications, we provide users with solutions that apply for a large scope of possible parameterization of applications.

Nevertheless, this approach features a few inherent drawbacks. One of them is the difficulty to model an exact behavior of the programs running on HPC facilities. As we previously discussed, supercomputers are very competitive environments that are constantly evolving depending on the applications currently running on the machine. For instance, the effects of I/O contention are very difficult to perceive, and even more difficult to foresee. Besides, hardware and system components variability is inevitable on large-scale machines. However, we believe that our models do not need to take all the specificities of applications and platforms into account in order to be efficient. They are tools that users can use to optimize the usage of the platform with their applications.

Moreover, models remain flexible tools that can always be enhanced, parameterized and adapted to a very wide range of use cases.

In this thesis, we propose solutions to optimize the performance of two different types of applications on large-scale machines :

Data-Intensive Applications: This category of applications includes many classical HPC workflows. Such applications are designed to run on a large scale, with possibly thousands of compute units. These applications are stable in terms of memory and computational needs, whatever parameters are entered. This means that for a given application, we can accurately estimate both the overall duration of the computations and their memory requirements. Another central feature is the date-intensivity they create. As we already mentioned, data management is critical in HPC systems and represents one of the major issues of this type of application. In this manuscript, we propose to use the paradigm called *in situ*, which consists in post-processing data directly where they are produced, in order to avoid the storage of the full set of data. The final volume to be written on disks is in that case very limited since it will only consist in the meaningful results generated by the application workflow. As they are stable in terms of resource requirements, we provide models to distribute the different tasks to the machine resources so that the execution of the application is minimized in time. The study of these applications constitutes the core of the first part of this manuscript.

Stochastic Applications: This new generation of applications arise from emerging fields using HPC facilities such as AI and neuroscience. Contrary to data-intensive applications, and because of the AI frameworks they use, these applications are input-sensitive. Indeed, they show important variations in terms of execution time (called walltime), depending on the input. Originally developed for local clusters, these applications have begun to target the computational power of supercomputers. However, the variation of resource needs is a major difficulty when one wants to run such programs on HPC facilities. Indeed, a user must provide a duration when he requests resources from the machine. With unpredictable variations of several hours, coming up with an estimation is difficult. An overestimation leads to a waste of resources, coupled with important expenses for the users. Conversely, underestimating the duration means that the applications will be stopped in the middle of its computation because the actual duration of the execution is greater than the one conveyed to the scheduler. The second part of this manuscript provides some important contributions aiming to determine reservation strategies that such applications could use. While the walltime variations can be important, we assume that these variations obey a known probability distribution that could be determined based on the previous runs of the application.

Outline

Let us now introduce the organization of this manuscript.

Part I contains our contributions to the issue of managing data-intensive applications on large-scale facilities. Chapter 1 introduces the scientific context of the topic under study. Chapter 2 proposes a review of the related work related dealing with the management of such applications on HPC facilities and with the *in situ* paradigm that we will use in this work to process such applications. Chapters 3 to 5 are dedicated to the presentation of all the solutions that we suggest to the problem under study together with an evaluation of our strategies. Finally, Chapter 6 concludes this first part of the manuscript and proposes some short-term perspectives for the presented contributions.

Part II hinges upon the study of stochastic applications. Chapter 7 introduces the inherent problems induced when managing such applications on HPC facilities by exposing our high-level observations based on the study of a representative application. Chapter 8 discusses some related work, while Chapters 9 to 10 describe reservation strategies that leverage the constant uncertainty about the execution time of this category of applications. Chapter 11 centers upon the performance evaluation of the different strategies in comparison with state-of-the-art approaches. Finally, Chapter 12 proposes to move one step further in the study of these applications since it features an in-depth profiling of a neuroscience stochastic application. We then demonstrate that a good knowledge of these applications is crucial to design and achieve cost-efficient strategies. Finally, Chapter 13 contains a summary of all contributions of this second part of the thesis.

This manuscript ends with a conclusion that summarizes the different contributions that have been brought throughout this thesis and mentions the publications that they were released in. We also propose ideas for mid-term to long-term future works that would allow to develop the different solutions described in this document.

Different appendices are also available in order to provide additional content when necessary.

Part I

Resource Management for Data-Intensive Applications at Scale

Chapter 1

Introduction

Contents

1.1 With high computing capacities come huge amounts of data

Many scientific fields study complex phenomena happening in our environment with the aim of better understanding them in order to answer various questions that have not been answered yet. For instance, the formation of our galaxy is subject to many different hypotheses that only the understanding of dark matter can clarify.

Most of the time, such studies are difficult to reproduce under laboratory conditions. For instance, the dark matter experiments are extremely difficult to perform in practice. Among the limitations are usually the overwhelming expenses they induce and the physical impossibility of doing it.

To overcome these difficulties, scientists build representative and expressive models that they use to perform physical or numerical simulations. Such experiments can validate, deny, or give some hints about the correctness of a hypothesis. These simulations are performed by using large-scale supercomputers.

Future supercomputer architectures are expected to reach a computing power in the order of an exascale [6, 7], which means more than 10^{18} FLOPS. Such a tremendous capacity would allow applications to solve extreme-scale scientific problems that have not yet been solved due to machine limitations.

While computing power keeps rising, some other features of supercomputers are not able to follow this trend. We already highlighted in the introduction to this manuscript the importance of data management. Moving data to the central memory of the machine becomes more and more critical to maintain system performance. While the computing capacities of machines are getting higher, the I/O capabilities of systems do not increase as fast. From [15], the peak performance of machines has been multiplied by a factor 500 from petascale to exascale machines. However, the I/O capacities have been increased by a factor ~ 70 at best, and the I/O bandwidth by a factor only ~ 30. This means that we are able to generate more data but unable to manage them efficiently due to the limited capacities of the I/O system, both in terms of access to memory spaces through the bandwidth and of storage capacities. Not only the bandwidth is limited, but there is also a large variability in I/O performance operations for applications. This is due to the high competition for system resources between users concurrently running their programs on the machine. Hence, several users may want to perform I/O operations at the same time, thus requiring to share the bandwidth with each other.

Altogether, I/O operations are most of the time blocking, thus hindering the applications. Hence, the overall performance of the applications is impacted, and this contention around I/O system has become critical for the performance of data-intensive applications. It follows that data management is nowadays a crucial concern for HPC applications running on supercomputers. This forces the HPC community to design solutions to help manage the tremendous amount of data generated by the applications.

One of these solutions is to add new memory spaces close to the compute nodes that are used as buffers. These buffers use persistent and fast storage technologies (for example Solid-state Drive (SSD) or Non-volatile Random Access Memory (NVRAM)) so that when a contention appears for a user, data are first stored on these local buffers. Afterwards, the application can continue its computation and maintain its performance. Once contention on I/O system is reduced, the buffers are emptied onto the PFS. In brief, this strategy allows users to reduce the impact of contention on application performance.

However, this approach does not allow to handle the huge amounts of data released at high frequency by data-intensive applications. As we previously explained, limited traffic bandwidth from compute nodes to PFS slows down the application which must then wait for all the data to be safely stored on PFS before starting a new iteration of simulation. Under these circumstances, limiting the requests to the Parallel File System (PFS) becomes necessary. To address this issue, new strategies are being developed that rely on coupling simulation and analysis in order to solve the problem of storing all simulation data. In the next section, we will describe this paradigm called *in situ* processing.

1.2 Description of *in situ* processing

Let us first describe in more depth the organization of a typical data-intensive application. HPC applications generally comprises two different phases. The first one is called simulation. It is a compute-intensive phase that can be envisioned as a single task involving a huge monolithic block of code that generates a large amount of data. The second one is the analysis (also called analytics), a data-intensive phase that consists in processing the data generated in the previous phase. This second phase aims at analyzing [106, 105, 31] or visualizing [56, 148, 75] data in order to extract insights from the phenomenon under study. Analytics is composed of different functions that we call analysis and considered as independent from one another. Simulation runs over several iterations, and analytics is periodically used to process the data. Some of the analysis functions can be run at each iteration of simulation, or after a predefined number of steps. In the end, analytics post-process the simulation data of iteration i when simulation is performing iteration $i + 1$. This means that the very first iteration of the application only runs the simulation, while the very last one only performs the analysis of the last simulation iteration. In between, simulation and analytics run at the same time. This system implies that the nodes are able to store the data of two iterations of simulation: the current one and the previous one, which is being post-processed by analytics. Overall, a data-intensive application is composed of many different tasks that one can divide into two main phases.

To limit I/O operations and the associated overheads, *in situ* has become one of the major paradigm to manage data-intensive applications. Its principle is to perform the analytics and simulation on the same machine and at the same time, without relying on intermediate files. Analysis is performed on-line, starting as soon as the data produced by the simulation are available in the memory of the compute nodes. Only the output of analysis functions is written to the PFS once all the analysis are finished. Thus, simulation and analytics share the same computing resources. The challenge is then to optimize resource allocation between the analysis and the simulation.

The *in situ* paradigm overcomes the limitations of this basic *post-mortem* data analysis by taking advantage of data locality and processing the analysis of the data directly where they are created. In essence, the *in situ* paradigm can be described as the fact of coupling the simulation and the analytics stages on shared (or not) resources.

First of all, let us briefly describe the structure of a typical compute unit. We focus here on CPU (also called processor or node), the basic brick of supercomputers as we presented in the introduction of this manuscript. Figure 1.1 features a schematic representation of an Intel Xeon E5-2680 v3 @ 2,5 GHz processor. It is composed of several cores (24 in our case), which are the components that perform computations following the VonNeumann architecture [145]. A processor also has memory spaces that can be accessed by the cores during computation. The principle of these memory spaces is that the closer to the core, the faster the access but the smaller the capacity.

Moreover, some spaces are shared between the cores (thus, possibly between users on the same processor) while some are dedicated to specific cores. Each core can actually access dedicated memory spaces called L1 and L2 caches. These storages have a limited capacity but can be accessed very fast by the cores. A third level of cache, L3, is shared by all the cores, and offers a bigger storage capacity. Finally, each processor has a non-persistent Random Access Memory (RAM) with quite an important storage capacity compared with the cache capacities (128GB for the whole processor in our case).

This RAM memory is used to store computational data during the application process. However, the RAM is not a persistent storage. Thus, users must transfer their data from the local RAM of processors to the disks of the PFS in order to save them once the application is finished.

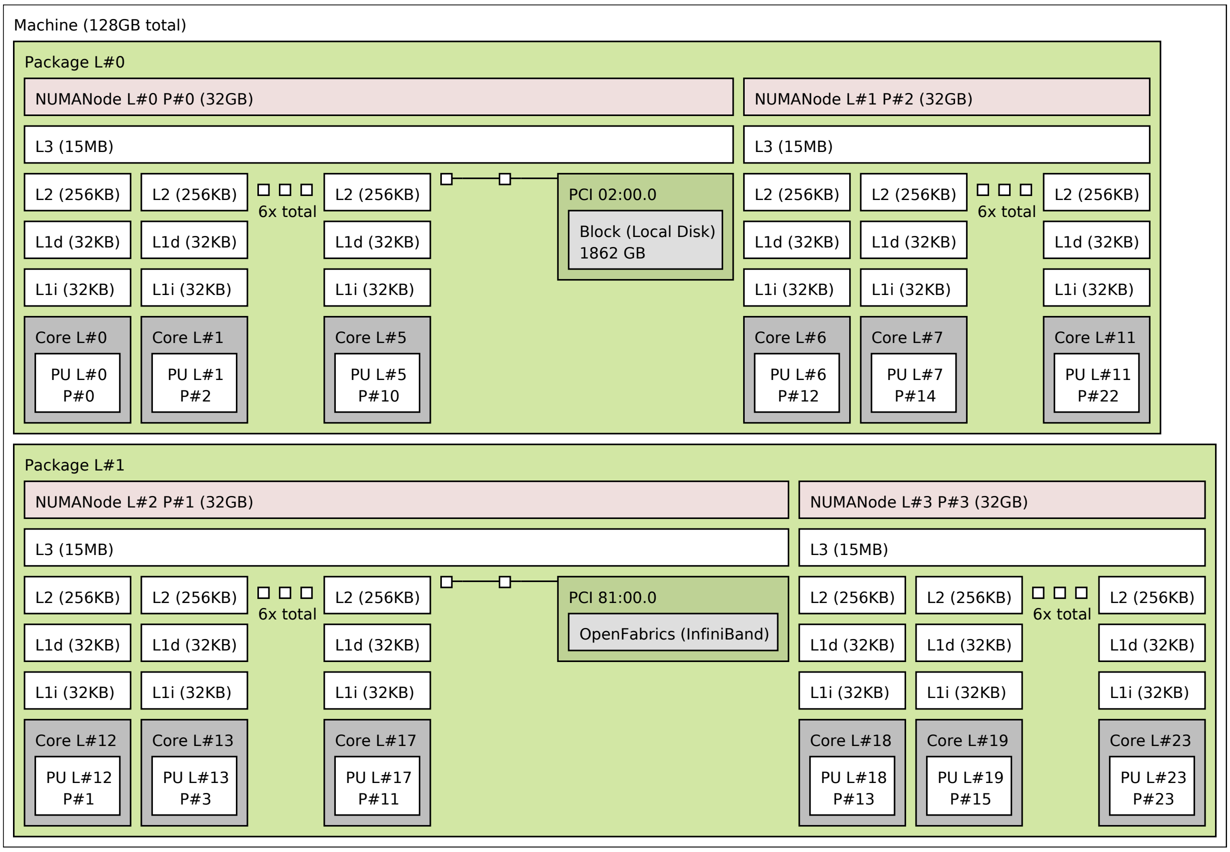


Figure 1.1: Features of a Haswell Intel Xeon E5-2680 v3 @ 2,5 GHz processor with 24 cores and 128GB of local memory represented by hwloc library [33]. Each cache level is denoted as L_n, and are increasingly shared following the value of n. The cache levels $L_n i$ are cache for instructions, while the ones with $L_n d$ are for data. I/O operations outside of the node are performed through Infiniband network.

Generally speaking, *in situ* processing aims at coupling simulation and analytics on machine resources in order to directly post-process simulation data, only the final analysis results needing to be stored on a disk. We will now detail the different ways of processing analytics alongside simulation on compute units. All these different possibilities form the *in situ* paradigm.

Analytics are called *in situ* if they are running on some cores of a node whose other cores are performing simulation. This principle is illustrated by Figure 1.2. In this example, on a node composed of 8 cores in total, 6 cores are dedicated to simulation while the last two are assigned to analytics. In such a configuration, the set of cores of the analytics are called *helper cores*. The advantage of this configuration is that the analytics can locally access the simulation data on this node. Most of the time, only few cores are diverted from simulation to prevent any major degradation of the simulation performance. Moreover, simulation often has difficulties scaling efficiently on a many-core approach. Hence, losing a few cores per nodes by removing them during the simulation is inconsequential. In that case, the *in situ* analytics are a good way to process data online. However, difficulties may appear such that negative interferences between the two phases. For instance, if one helper core begins trashing the L3 cache, it may have a dramatic effect on the performance of other cores.

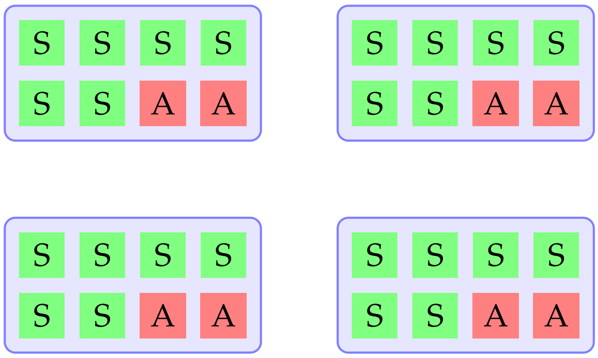


Figure 1.2: *in situ* processing of analytics where simulation and analytics run in parallel on 4 processors of 8 cores. Simulation runs on 6 cores while the analytics use the 2 other cores of each node.

Another way of coupling simulation and analytics is to dedicate nodes to each of the phases. This is named *in transit* processing. Figure 1.3 illustrates such situation. Simulation and analytics are coupled on separated nodes. In this setup, analytics benefits from the full resources of its assigned nodes. These nodes are called *staging nodes*. In this configuration, simulation and analytics run on separated resources, which prevents any interference from happening between the two of them. However, the main difference is that staging nodes do not have simulation data in their local memory. Hence, it is necessary to transfer the required data from the simulation nodes to the staging nodes in order to post-process them. As we already discussed, I/O operations can face contention that can impact the performance of the staging nodes.

Finally, *in situ* and *in transit* can be combined in a hybrid setup, as illustrated in Figure 1.4. This is a flexible approach in which analytics that do not scale correctly can be performed *in situ*, while the most compute-intensive of them that would interfere too much with simulation can be exported towards staging nodes.

Figures 1.2, 1.3 and 1.4 assume that simulation and analytics run in parallel on their assigned cores. This type of execution is called asynchronous processing. There is another version called synchronous where simulation and analytics run by successive iterations on the *in situ* nodes. Simulation first runs on *in situ* nodes using all of their cores. Then, the simulation is periodically stopped to perform the analytics on the same

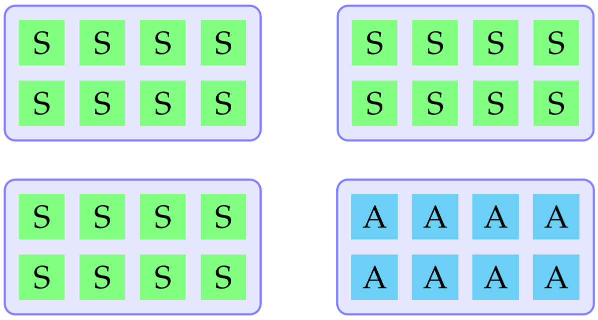


Figure 1.3: *in transit* processing of analytics, during which simulation and analytics run in parallel on separated processors of 8 cores. Simulation runs on 8 cores and on 3 processors, and the analytics on 8 cores of one processor.

resources. Chapter 3 contains a modelization of both execution types and discusses the main advantages and drawbacks of both schemes.

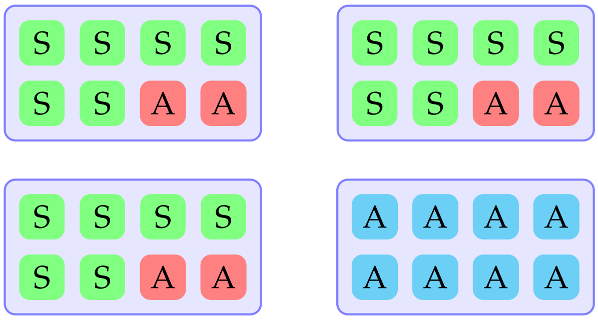


Figure 1.4: Hybrid *in transit*/*in situ* processing of analytics on 4 processors of 8 cores. Simulation runs on 6 cores and the *in situ* analytics on 2 helper cores of 3 *in situ* processors. *in transit* analytics are executed on a dedicated processor.

There are several software solutions [50, 53, 52] that allow users to specifically dedicate nodes and cores to analytics and distribute the computation tasks over different sets of nodes. Thus far, they rely on a manual resource partitioning and on the allocation by the user of task requirements (simulation and analysis). This means that they require an important human effort to efficiently match the application needs with the machine resources. Moreover, this approach will not necessarily be appropriate for another one.

This part of the manuscript features several contributions to automate the arbitration of resources between simulation and analytics. We propose a memory-constraint modelization for *in situ* analytics. We use this model to provide different scheduling policies and determine both the number of resources that should be dedicated to each analysis function, and strategies to schedule efficiently these functions.

1.3 Contributions

In this part, we propose to optimize data-intensive applications on supercomputers through a theoretical approach. We first design flexible models of HPC infrastructures and of the applications that are running on it. Secondly, we show the general application pipeline that these models can express. From this, we formalize an optimization problem and identify the critical resource arbitration and scheduling problems to which we propose practical solutions. Finally, we evaluate our solutions through an intensive set of simulations on synthetic applications.

Overall, we make the following contributions:

- We propose a general model for HPC applications, which will take into account simulations and analysis functions as well as propose a model for the target machine, including multi-core architectures, memory limitations, and communication costs under bandwidth constraints. We use it to properly define the *in situ* allocation problem.

- We use this model to derive new algorithms for the allocation and scheduling problem. The algorithms are based on a theoretical analysis of the model followed by greedy strategies. We also provide an optimal (non polynomial) solution which will be used later to study our strategies.

- We evaluate these algorithms on synthetic applications. This evaluation allows us to point out the key elements to take into account when scheduling *in situ* functions.

We insist on the fact that the model that we study supports mixed strategies for the analysis scheduling. The analysis can be performed on the same nodes that run the simulation, either in sequence with the simulation execution or in overlap following a time or space sharing strategy. These strategies are commonly referred as *in situ*. But they can also be performed *in transit*, i.e. on nodes dedicated to the analysis and which take into account the extra cost of the data transfer from the simulation nodes to these staging nodes. Therefore, we believe that this work can be used for any data-intensive application running on supercomputers.

Now that we have introduce the issue under study in this first half of the manuscript, let us explain how we structured the rest of this part in order to resolve the issue of resource management for data-intensive applications.

Chapter 2 introduces some related work about scheduling applications on HPC facilities. We also present data management techniques in HPC, so as the related work about *in situ* paradigm for HPC applications.

In Chapter 3, we present the application and platform models for *in situ* processing. The platform model includes task representation and communication model. Application model formulates a general representation of the two phases of data-intensive applications by describing a task with an execution time on a single core and a peak memory requirement. We also introduce the task performance model on shared resources. Finally, we describe how to combine platform and application models in order to describe the application pipeline in both synchronous or asynchronous execution.

Chapter 4 formulates the optimization problem that comes up when we try to minimize the application processing time on HPC infrastructures. After enlarging upon this issue and its states, we propose to solve this problem by decomposing it into two subproblems, each receiving a proper solution. The first sub-problem consists in determining the nodes and cores partitioning between simulation and analytics when analytics are already assigned to be processed either *in situ* or *in transit*. We base our solutions to this sub-problem on two different task performance models to show the flexibility of our proposition. The second sub-problem is then to determine which analysis have to be performed *in situ* and eventually on staging nodes. We propose scheduling heuristics in combination with solutions to the first sub-problem to solve the overall optimization problem.

Chapter 5 proposes an evaluation of our solutions on synthetic applications. We evaluate both asynchronous and synchronous execution schemes and evaluate our scheduling heuristics by different metrics. From the results, we extract the main criteria for application performance, the most important one being the correlation between application performance and performance of helper cores dedicated to perform *in situ* analytics.

Finally, Chapter 6 presents a conclusion to this first part of the manuscript. We also provide different perspectives to this work, and introduce content of the second half of the manuscript.

Chapter 2

Related Work

Contents

In this chapter, we present an overview of the related work concerning the management of data-intensive applications on HPC facilities. We also propose a state-of-the-art summary of the *in situ* paradigm in the context of supercomputers.

2.1 Running applications on supercomputer facilities

This section outlines how to run an application on HPC facilities.

Submitting an application onto HPC platforms consists in submitting a request to a complex program called scheduler. In their request, users must specify the amount of resources needed (number of nodes/cores as well as optionally the type of nodes and/or the amount of memory per core required by the application). Users must also provide a total runtime for each submitted job. The scheduler takes all this information into account when setting up the job priorities as well as when choosing the set of nodes involved in each execution. Concerning the size of the reservation, data-intensive applications have a rather constant execution time on a determined set of resources. Hence, users are able to accurately determine the size of the reservation without overestimating it. This estimation is not always feasible to perform for other profiles of applications, as we will see in the second part of this manuscript.

Batch schedulers are widely adopted by many resource managers in HPC systems, such as Slurm [155], Torque [131] and Moab [37]. They use an iterative and repetitive algorithm triggered by state changes, such as new job submission, job starting or ending, or timeout. They use different policies to determine which job should be executed when and on which resources. Jobs are usually placed in one or multiple waiting queues with different priorities before being scheduled onto the machine. For example, the Slurm scheduler [156] uses two queues: one for high-priority jobs, and another for low-priority jobs. Jobs are placed in queues based on their resource requirement, generally with long-running jobs that require a large amount of resources having higher priorities in order to avoid starvation. Jobs that are kept in the waiting queue for a long period of time could also be upgraded and moved up in the queue. HPC scheduler has to balance a continuous trade-off between system efficiency (increase machine usage by scheduling large jobs) and application response (the time small jobs need to wait in queue before being executed). Slurm [155] schedules the jobs by starting with the top of the high-priority queue, and then moving down. In addition, most batch schedulers use resource reservation in combination with backfilling [98, 129, 114]. Backfilling allows smaller jobs lying farther back in the queue to be scheduled before the other jobs waiting at the front of the queue, as long as the jobs at the front are not delayed by this decision. Even though larger jobs (in terms of time and resource requirement) have higher priorities, the lack of resource availability in the system generally leads to longer waiting times. On the other hand, smaller jobs, despite having lower priorities, are usually scheduled quickly thanks to the backfilling algorithms that place them in the unused time slots between successive large jobs.

Some studies (e.g., [152, 113, 124]) have analyzed the impact of scheduling strategies on the performance of applications in large HPC centers. They show that the penalty for jobs with longer requested walltimes and/or larger numbers of nodes is higher than that for jobs with shorter elapsed times and smaller number of nodes. This can be observed, for example, on the K computer from Riken Advanced Institute for Computational Science [152]. The HPC scheduling policy generally tries to give users a fair opportunity for job execution while maximizing the total system utilization. The study shows that, for applications requesting similar computing resources, the waiting time generally increases with larger requested processing times which can cause hours of delays for large scientific applications, even though it also depends on other workloads submitted to the system. Some HPC centers divide the resources into seasons for users to utilize the allocated resources. Users tend to submit more jobs toward the end of a season, causing contention at the scheduler level that results in even longer waiting times. In addition, depending on the overestimation of the required processing time for long jobs, smaller jobs might have more opportunity to be scheduled through the backfilling algorithm. Medium jobs have lower priority than large jobs, without being candidates for the backfilling algorithm. Thus their waiting time may be quite significant depending on the workload of the HPC system. The study retrievable in [124] presents an evolution trend of the workload of HPC systems and its corresponding scheduling policies as we move from monolithic MPI applications to high-throughput and data-intensive jobs. The cost paid in terms of waiting time of applications in the queue has generally increased over the years because of the fluctuating workloads. Systems that give each job a partition of the resources for exclusive use and allocate such partitions in accordance with the sequence of job arrivals could suffer from severe fragmentation, leading to low utilization [113]. The authors of the research propose an aggressive backfilling algorithm for dealing with such fragmentation. However, users are still expected to provide accurate runtime estimates. The study shows that over-estimations may lead to a long waiting time, and possibly to excessive CPU quota loss, while under-estimations can lead to job terminations before their completion. Some recent schedulers [116] consider the distribution of execution time of the submitted jobs to take their scheduling decision in order to increase their overall utility.

Finally, BigData frameworks such as MapReduce [42] or Dryad [84] also rely on schedulers (e.g. YARN [143] and Mesos [77]) considering distinct features such as fairness or resource negotiation to manage the workload.

2.2 Emergence of *in situ* as a major paradigm

Data management has become a major issue in the field of supercomputers. We already mentioned the possibility to use burst-buffers to temporarily store data if the I/O system is congested. Once the contention is reduced, buffers are emptied on disks. From an application perspective, performance has not been degraded and no data loss has occurred. A few researchers have investigated the issue of buffer size and how to partition them between different users [22, 21]. Strategies to efficiently empty burst buffers have also been proposed [135]. Indeed, draining burst buffers data must be done with care in order to avoid creating even more severe I/O contention. What is more, some studies [99] have attempted to show the efficiency of burst buffers in large-scale storage. Finally, some other works have proposed to use these burst buffers as storage places for input data rather than for output data [154]. The goal is to optimize BigData frameworks on HPC facilities. The authors' idea was to prefetch data on these buffers in order to have fast access to meaningful data for applications.

Obviously, such solutions are not sufficient due to the limited size of burst-buffers. Nowadays, a wide range of scientific domains such as biology, chemistry, or dynamic system studies use large scale simulations. As a consequence, the *in situ* paradigm emerged and progressively became one of the major solutions for data management in HPC. Many large scale simulations have developed analytics techniques that we will discuss below.

Solutions to *in situ* visualization and analysis are usually described by where the process is performed and on how resources are shared. This has led to the common distinction between *in situ*, *in transit*, and postmortem visualization, as well as between tight and loose coupling. To alleviate the I/O bottleneck, the *in situ* paradigm offers the possibility to start processing data as soon as they are made available by the simulation, while still residing in the memory of the compute node. *In situ* processing includes data compression, indexing, and computation of various types of descriptors (1D, 2D, images, etc.). The amount of data to be saved on disks is reduced, hence reducing the pressure on the file system when writing the results, as well as when reading them back for the postmortem analysis. Results can also be visualized on-line, enabling advanced execution monitoring.

Per se, the idea of reducing data output to limit I/O related performance drops and to keep the output data size manageable is not a new one. Scientists have relied on solutions as simple as decreasing the output frequency. *In situ* processing proposes to move one step further by providing a full fledged processing framework that enable scientists to manage more easily and thoroughly their available I/O budget. The first publication to have ever coined the *in situ* concept in those terms is very likely the one written by Kwan Liu Ma et al. [103]. Solutions emerged from existing visualization libraries like VTK or Visit that added new readers to get data directly from a live simulation [56, 148], or from I/O libraries like ADIOS [101] augmented with processing capabilities when transiting data from the simulation to the file system [162]. While users have to configure data placement in software, none of them are able to determine if this placement is optimal or not.

The most direct way to perform *in situ* analytics is to inline computations directly in the simulation code. This is the approach adopted in [158] as well as in the standard visualization tools like Paraview and Visit [56, 148] and their recent extensions [25, 93]. In this case, *in situ* processing is executed in sequence with the simulation that is suspended meanwhile. This process is described in Figure 2.1. We name this approach *synchronous*.

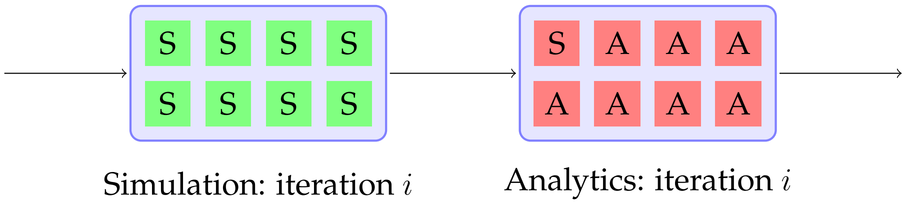


Figure 2.1: Synchronous *in situ* processing of analytics where simulation and analytics iteratively run on a processor of 8 cores. Simulation runs on the 8 cores and is then paused to perform the analytics on 8 cores.

In order to improve resource usage, *asynchronous in situ* proposes to overlap the simulation and the analysis executions, as illustrated in Figure 2.2. At each iteration of simulation, some cores of *in situ* nodes are allocated to the simulation, while the remaining nodes are running the analytics. At iteration $i + 1$ of simulation, analytics processes the data of the iteration i of simulation. This implies that the data of two iterations of simulation are stored on compute nodes. An obvious solution to manage resources distribution consists in relying on the OS scheduler capabilities to allocate resources. The analytics runs its own processes or threads concurrently with the ones of the simulation. The simulation only needs to give a copy of the relevant data to the local *in situ* analytics processes. The analytics can next proceed concurrently with the simulation. However, some works [161, 72] show that relying entirely on the OS scheduler does not prove to be efficient because the presence of analytics processes tends to disturb the simulation (context switch, cache trashing). GoldRush proposes to activate analysis executions only on long-enough sequential sections of the simulation. Tins adopts a task-based programming relying on a work-stealing scheduler for a fine grain interleaving and load balancing of tasks on the compute nodes [46].

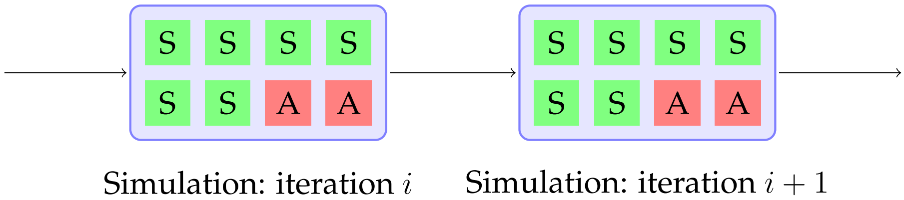


Figure 2.2: Asynchronous *in situ* processing of analytics where simulation and analytics run simultaneously on a processor composed of 8 cores. Simulation runs on 6 cores and the analytics working on the data of previous iteration of simulation has 2 dedicated cores.

To reduce these interferences, several works propose to rely on space sharing where one or several helper cores are dedicated to analytics. Damaris [50], FlowVR [53], Melissa [139], Functional Partitioning [96], GePSeA [127], Active Buffer [104], or SMART [146] adopt this solution. Tins [46] introduced dynamic helper cores, dedicating cores to analysis only when analysis tasks are ready to be run. Even if the *in situ* processing simply consists in saving data onto disks, this approach tends to be more efficient than the one consisting in relying on standard I/O libraries like MPI I/O [50]. The simulation runs on less cores, but, since it is usually not 100% efficient, the performance is decreased by less than the ratio of confiscated cores. Still, helper isolation is limited to the compute core and the first two cache levels. Memory and usually the L3 cache are shared between cores. *In situ* tasks, by trashing the L3 cache or using a significant amount of the node memory, can affect the simulation performance. Communication that can perform *in situ* tasks can also load the network interface and slow down the simulation communications. To better schedule the communication load, DataStager [12] proposes a mechanism that triggers *in situ* related traffic outside of the simulation communication phases.

For a better isolation of the simulation and *in situ* processes, one solution consists in offloading *in situ* tasks from the simulation to the costs of moving the data from the simulation nodes to the staging nodes. This way of processing is called *in transit*. HDF5/DMS [31] uses the HDF5 and PnetCDF I/O library calls to expose a virtual file to staging nodes. GLEAN [141, 144] is another example of a simulation/visualization coupling tool initiated by making HDF5 and PnetCDF I/O library calls. DataSpaces [48] stores the simulation data on staging nodes with a spatially coherent layout and acts as a server for client applications. Padawan [38] proposes a Python based staging solution. Several systems use staging nodes to expose the simulation data to other scientific workflows [43, 102].

A few frameworks support both *in situ* and *in transit* processing. JITStaging [11] and PreData [163] propose to extract data from the simulation, apply a first *in situ* treatment with simple stateless codes, and then transfer the data to staging nodes. Bennett et al. [30]'s solution is built on top of DataSpaces and Dart [47] to perform *in situ* and *in transit* visualization and analytics. FlexIO [162] brings to ADIOS *in situ* and *in transit* processing capabilities. FlexIO uses shared memory segments to handle data to asynchronous node-local *in situ* processes and RDMA transport methods for inter-node transfers, in particular for staging nodes. Specific stateless codelets can be dynamically moved on different cores during the simulation. For $N \times M$ like data re-distributions, FlexIO relies on centralized coordinators that gather information about data and process distribution, compute the communication pattern, and send back the necessary instructions to each process involved. This handshaking process can be totally or partly bypassed if the data distribution does not need to be recomputed between consecutive steps. Zhang et al. [159] added a shared memory space to a Dart server to support both simulation code coupling and *in situ*/*in transit* scenarios. The user describes groups of parallel codes called bundles and creates a workflow between these bundles. Based on MPI, the framework requires that all bundles are integrated in the same MPI application, which can require significant coding efforts. Similarly, Damaris [52, 50] proposes to embed *in situ* tasks in the MPI context of the simulation. At launch time, MPI processes define the type of task that they are about to execute (simulation or *in situ*) depending on their mapping on the target machine. Then, cores or nodes can be dedicated to either *in situ* or *in transit* tasks. Bredala [54] enables automatic global data redistribution between the *in situ* and *in transit* nodes and provides a contract that enables to extract data from the simulation depending on the analysis demands [112]. Having a single MPI application seems quite interesting for those supercomputers OS that do not support running more than one application per node. FlowVR [53] relies on a dataflow model in which components can be mapped *in situ* or *in transit* without the constraint of having all components running in the same MPI context.

Most of these approaches are MPI+X based. New programming models are also developed as alternatives to message passing. StarPU [20], PaRSEC [78], Legion [26], and HPX [86] propose task-based runtime systems for distributed heterogeneous architectures. The program defines a directed acyclic graph where vertices are tasks and edges represent data dependencies between tasks. The runtime is in charge of mapping tasks to resources, triggering task execution, and deciding of the necessary data movements once data dependencies are resolved. Early experiments have attempted to utilize Legion for *in situ* analytics [119, 74]. Though they show that Legion runtime is able to overlap analytics and simulation tasks, there are also evidence that the performance of Legion is not yet as competitive as MPI approaches.

Very few works have addressed *in situ* application modeling to define algorithms, study scheduling policies, and manage resource partitioning. One of this work is [94] that proposes a statistical performance model based on algorithmic complexity to predict the run-time cost of a set of representative parallel rendering algorithms. These are commonly used for *in situ* rendering, yet they do not take into account the interactions between the simulation and the analytics components. Another group of works based on DataSpace focuses on optimizing *in transit* data storage. Some studies [133] focus on performance optimization through data placement. Stacker [132] relies on machine learning to optimize the placement of data on the memory hierarchy, taking into account per node persistent storage capabilities like NVRAM or Burst Buffers. Zheng et al. [162] also put forward several heuristics enabling to compute the core mapping and to optimize the use of helper cores and staging nodes. Furthermore, Malakar et al. [106, 105] considered *in situ* analysis as a numerical optimization problem to compute an optimal frequency of analytics when the later is subject to resource constraints such as I/O bandwidth and available memory. However, this work only offers sequential simulation and analysis.

Chapter 3

On Modeling Applications on HPC facilities for *in situ* Processing

Contents

In this chapter, we present mathematical models to schedule a HPC application composed of a simulation code and the associated analytics on a HPC infrastructure. Section 3.1 introduces a model to abstract HPC machines, while Section 3.2 presents the application representation, that is based on the assumption that the target applications generate lots of data that can be analyzed on-line. Finally, Section 3.3 presents the application pipeline that combines the models above introduced.

3.1 Architecture

In this section, we introduce a model of the machine infrastructure.

We assume that a platform is composed of C_n identical unit-speed nodes. Each node is composed of c cores (also called processors throughout this work) and a shared memory of size M_n between all the cores.

We model communications as an extra cost. Intra-node communications are considered as cost-free due to the shared memory space between processors. We assume that a processor can access the data on its node memory without extra cost. However, inter-node communications generate an overhead that is modeled as follows.

We define a communication cost $V \mapsto T_{\text{com}}(V)$: assume a volume of data V located on the memory of node N_i, then it takes $T_{\text{com}}(V)$ units of time to transfer it on the memory of node N_j.

In this work we use the linear bandwidth model [149] to model the communication cost[1]:

$$T_{\text{com}}(V) = \frac{V}{b}$$

where b is the available bandwidth for the transfer. We assume that total bandwidth of the system is equally divided between each node, as is the total memory for each node.

Note that other classical communication models take into consideration a latency that fades-out for large messages. We consider here significant data movements, hence this latency has a negligible effect in our case.

3.2 Applications

This section is devoted to the application representation.

This work focuses on iterative HPC applications that consist in two different phases: the simulation phase, a compute-intensive phase modeled as a huge task that generates large amounts of data, and the analysis phase (also called analytics), a data-intensive phase that consists in transforming the data generated in the previous phase. Analytics is composed of different functions that we consider as independent from one to another. Thus, many different tasks have to be modeled to represent such double-phase applications.

Our model supports both *in situ* and *in transit* processing of the analysis tasks. Part of the analysis phase can be executed *in situ* on the nodes used for the simulation either in a synchronous or asynchronous mode. The cores of the nodes dedicated to analytics in asynchronous setup are the helper cores. The other part runs *in transit* on dedicated nodes, the staging nodes.

3.2.1 General representation of application tasks

We consider that all tasks, being either simulation or analysis tasks, are moldable: the scheduler can decide prior to execution how many nodes and cores are assigned to each task. Once the application starts, this number is fixed for the full computation duration.

The work of a task is defined as the execution time of the task times the number of cores used. A classical task model considers that the amount of work is constant with the increase in pluralization (*perfectly/embarrassingly parallel* model [76]). In other words, parallelizing the application implies no overhead. In the next chapter, we propose a solution for two different task models: perfectly parallel and Amdahl's law [17]. However,

[1] Although, most of this work is agnostic of the bandwidth model used

to ease readability, we focus mostly on the perfectly parallel model for its expressivity. Most of the results are valid with many other work models, and when needed we discuss in this work the impact of such models.

Each task/function (simulation or analysis) is defined by two parameters:

- a reference execution time given as a running time on one core; and
- a memory peak representing the maximum memory consumption of the task during its execution.

For notation, we introduce $t_{\mu}^{\rho}(c)$ as the makespan of task μ (either simulation or analysis function) at iteration ρ on c core(s).

3.2.2 Simulation phase

We consider the simulation as a task iterated Π times. In the following, let S_ρ ($\rho \in \{1, \cdots, \Pi\}$) be the ρ^{th} simulation iteration.

Each iteration S_ρ is defined by its data input V_ρ, its execution time $t_{\text{sim}}^{\rho}(1)$ and its memory peak P_ρ.

Assuming there is enough memory available to perform the iteration S_ρ, let $t_{\text{sim}}^{\rho}(c)$ be the execution time on c cores to perform the simulation,

$$t_{\text{sim}}^{\rho}(c) = \frac{t_{\text{sim}}^{\rho}(1)}{c}$$

We also work under the assumption that the memory peak does not depend on the number of cores working on the iteration. Hence for iteration S_ρ running on c cores, the average peak memory per core is $\frac{P_\rho}{c}$.

We consider that the simulation data are evenly distributed amongst the nodes assigned to the simulation, and thus that the simulation outputs are also evenly distributed. We consider that the simulation does not perform I/Os directly. Simulation outputs are all handed to analysis tasks that are in charge of data processing and eventually to perform I/O to save the necessary data to disk.

3.2.3 Analysis phase

An analysis phase runs after each simulation iteration S_ρ. We denote by $\mathcal{A}^\rho$ the set of analysis tasks to be performed on the output data of iteration S_ρ:

$$\mathcal{A}^\rho = \{A_1^\rho, \cdots, A_{K^\rho}^\rho\}$$

Any of the tasks of $\mathcal{A}$ can be executed either *in situ* or *in transit*. We denote by $\mathcal{A}^{IS}$ and $\mathcal{A}^{IT}$ the partition of the analytics tasks according to their execution mode:

$$\mathcal{A}^{IS} \,\dot{\cup}\, \mathcal{A}^{IT} = \mathcal{A}$$

For this work, we assume that all analysis tasks run at each simulation iteration ρ are identical:

$$\mathcal{A}^{\rho} = \mathcal{A}$$

Analytics tasks can include about anything, like computation of high level descriptors [46], compressing data [138], verifying the integrity of the data to detect a silent error [23], or checkpointing for reliability [24].

Analytics tasks can either be executed synchronously, i.e. without overlap with the simulation or asynchronously. We detail both scenarios in coming paragraphs.

For now, we denote by $t_i(c)$ the processing time of function i on c cores.

***in situ* Analysis:** The *in situ* analysis tasks directly access the local simulation output data on the node where they run. As all tasks run in parallel on each simulation node, the total makespan of *in situ* analysis functions is the sum of the makespan of each task.

The *in situ* tasks need memory space allocated in the simulation nodes. We ensure that this does not impair the simulation execution by enforcing that the sum of the peak memory of both the simulation and analytic tasks running on the simulation node do not overflow the total memory of each simulation node.

If we perform synchronous analytics, the simulation is periodically paused to perform *in situ* analytics on simulation resources. It means that simulation and analysis alternatively use all the cores of their assigned nodes.

For asynchronous analytics, we used so called *helper cores* to perform the *in situ* analysis in parallel of the simulation. In this case, analysis and simulation overlaps. Because of helper cores, the simulation is slowed down and a trade-off between analysis benefits and simulation performance loss has to be addressed.

I/O are performed to transfer the output of analytics to the PFS. As it is assumed to have negligible cost, the cost is included into the function makespan.

***in transit* Analysis:** The input to the $\mathcal{A}^{IT}$ functions are not available on the *in transit* nodes. They need to be transferred from the simulation nodes.

We model the *in transit* cost in terms of time only. We assume that the cost is the time to send all the input of functions $\mathcal{A}^{IT}$ (i.e. all associated memory peaks) from simulation nodes to *in transit* nodes, associated to the cost for running the functions. The output of analysis functions are stored on PFS, which is assumed to induce a negligible cost that is not subject to interference. Thus, we consider that this cost is included in the makespan of the functions. We also consider that it is the *in transit* nodes that has to face the transfer overhead while the *in situ* nodes on which data are located can continue working.

In both asynchronous/synchronous scenarios, *in transit* analytics is performed on a dedicated set of nodes.

We touch here a first optimization problem that arises: if all analysis are performed *in transit*, required inputs have to be transferred to staging nodes, hence generating an overhead. However, if all analysis are *in situ*, simulation will be processed in parallel of analysis, thus will be slowed down.

Figure 3.1 illustrates the partitioning of the resources between *in situ* and *in transit* processing. Simulation runs on $c - c^*$ cores on n^* *in situ* nodes while the c^* other cores of these nodes are dedicated to *in situ* analytics. Finally, $C_n - n^*$ are staging nodes for *in transit* processing of analytics, for which the transfer of simulation data are necessary.

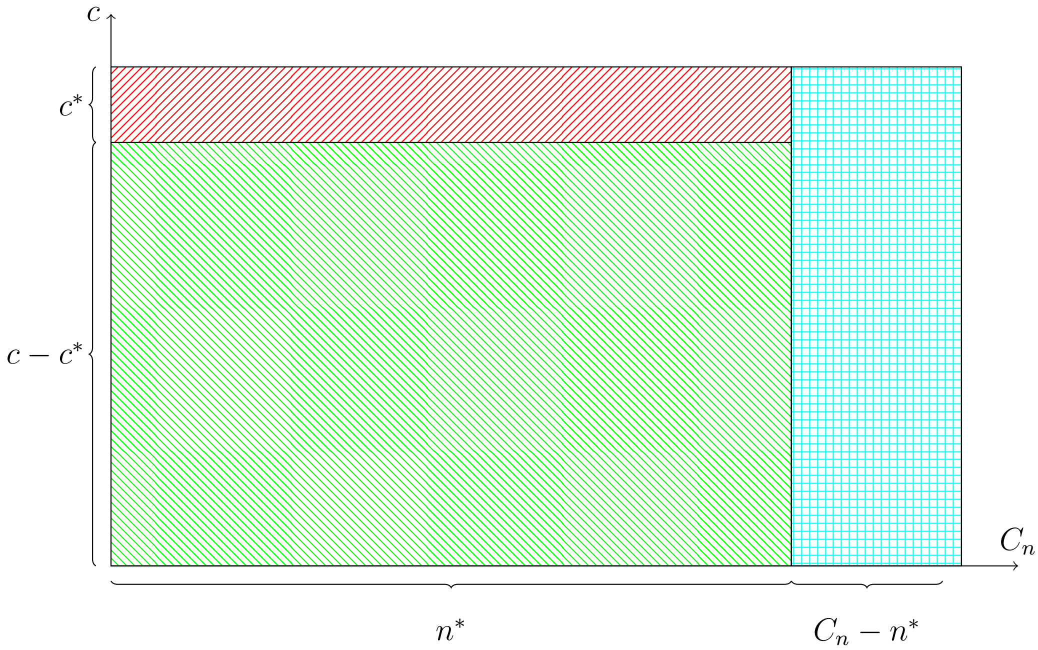


Figure 3.1: Schematic representation of resource allocation between *in situ* and *in transit* processing.The x-axis represents the number of nodes and the y-axis the number of cores on one node. Simulation cores are in green, *in situ* cores in red while *in transit* cores are in blue.

3.3 Application pipeline

We end this chapter dedicated to model design by the presentation of the application pipeline. We model both types of execution when coupling simulation and analysis: asynchronous and synchronous processing.

Figure 3.2 presents the application pipeline for asynchronous analytics.

In this model, the cores dedicated to *in situ* analysis cannot be used for the simulation. They are called helper cores. For each simulation iteration, the set $\mathcal{A}^{IS}[\rho - 1]$ is executed concurrently to the ρ^{th} simulation round on n^* *in situ* nodes. Note that they do not need necessarily to be executed concurrently, but this is one of the strategies that minimizes the execution time since the helper cores cannot be used for anything else that analysis.

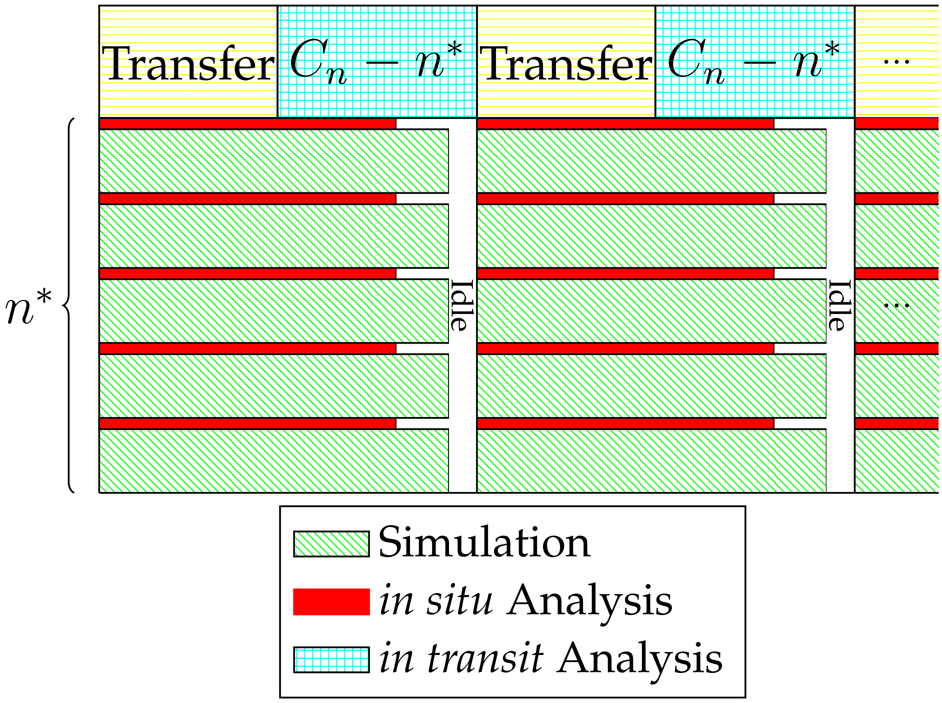


Figure 3.2: Illustration of Application Workflow with Asynchronous Analytics.

There are c^* helper cores over the c cores that are dedicated to perform *in situ* $\mathcal{A}^{IS}$ functions. Thus, $c - c^*$ cores are dedicated to the simulation iterations. The helper cores can access simulation data locally, without requiring to transfer the input of the analysis functions. However, it reduces the number of cores dedicated to simulation, and its performance. The set of *in transit* functions is performed on $C_n - n^*$ nodes, using all available cores. However, the transfer of function inputs induces a time overhead that has to be balanced with the makespan of simulation and *in situ* analysis. Finally, the application makespan is computed as the maximum time between simulation, *in situ* and *in transit* makespan. Some idle time on resources appears when one of these makespans is longer than another.

Usually, synchronous execution signifies that the analytics code is directly included in the simulation code. This is mainly used for analysis functions that periodically monitor the evolution of a given parameter or value of importance for the application progress. Heavier compute-load analysis can also be performed synchronously at a lower frequency and mostly accounts for reducing the final amount of data to be stored. One example of synchronous *in situ* processing is the Paraview toolkit [56] that enables users to embed visualization functionalities inside simulation code.

Figure 3.3 presents the application pipeline for synchronous analytics. The main difference with the previous scenario is that the *in situ* analytics is processed in sequence with the simulation, that is paused during that time. The *in transit* analytics follows the same rules as before. As for asynchronous case, some idle time may appear, for example

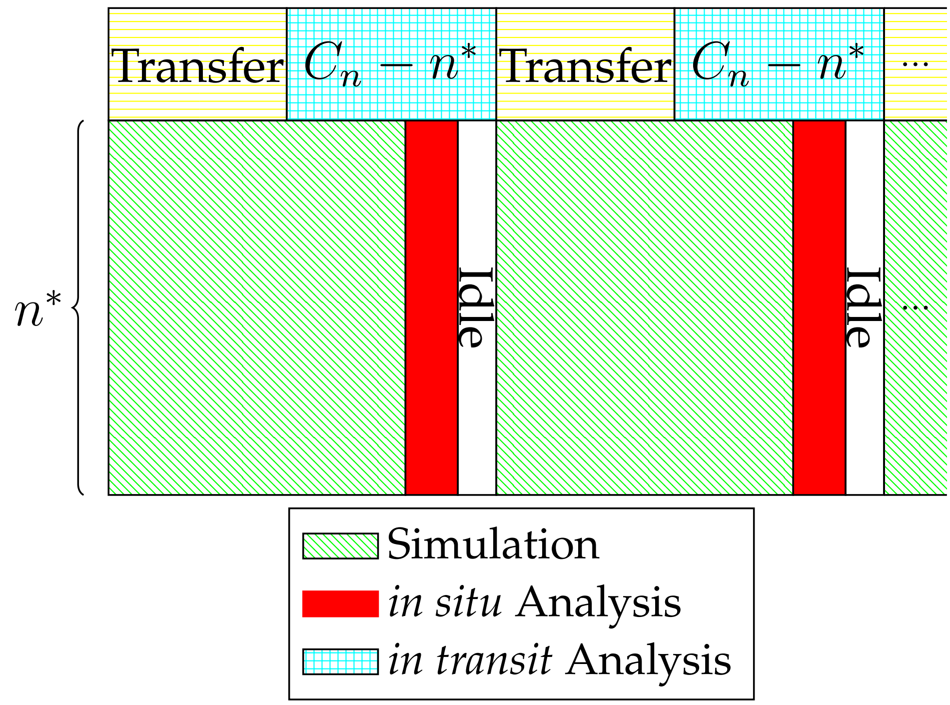


Figure 3.3: Illustration of Application Workflow with Synchronous Analytics.

if the *in transit* analytics has a larger makespan than the simulation followed by *in situ* analysis. In the synchronous case, the application makespan is the maximum between the simulation summed with the *in situ* analytics and the *in transit* makespan.

In the rest of this work, we assume that the analysis functions are performed using the asynchronous scenario. This can be justified by a difference in practice between the two scenarios. Indeed, synchronous analytics causes complex side effects such as cache pollution overheads during the switch between simulation and analytics on the *in situ* resources. Moreover, it is often intrusive to the code of the application, as we previously stated. Finally, another reason is that the simulation process is often not able to efficiently scale while we increase the number of cores. This is even reinforced when considering another task model than the perfectly parallel one. Hence, dedicating full resources of the *in situ nodes* to the simulation does not necessarily enhance its performance. However, in Chapter 5, we discuss the performance of both asynchronous and synchronous analytics to demonstrate the benefits of our contributions.

This ends the presentation of the models of the application and the platform. In the next chapter, we formalize the optimization problem under study and show how to derive its solutions.

Chapter 4

Automatic Resource Partitioning and Scheduling Heuristics for *in situ* Processing

In this chapter, we will express in Section 4.1 the optimization problem for *in situ* processing of data-intensive applications that we further decompose into two sub-problems. Sections 4.2 and 4.3 detail solutions for the first sub-problem for two different task models. Finally, Section 4.4 will introduce scheduling heuristics to solve the second sub-problem.

4.1 A global optimization problem for *in situ* processing

In this section, we express the different problems to be solved in order to optimize HPC applications on HPC facilities.

Firstly, we define the general optimization problem. Recall that we process all analysis functions at each iteration of simulation. We now propose Problem 1, called 1-ARP (1-APPLICATION RESOURCE PARTITIONING):

Problem 1 (1-APPLICATION RESOURCE PARTITIONING (1-ARP))**.** *Given $\mathcal{A}$ the set of analysis tasks, $\mathcal{S}$ the set of simulation tasks, can we determine n^* the number of simulation nodes, c^* the number of helper cores and a scheduling of the N calls to the set $\mathcal{A}$ such that the total makespan of the application is minimal?*

This is the general problem we will solve in order to get an efficient distribution of application workload over an optimal distribution of computational resources.

In this problem, we can identify two subproblems with regards to our platform and application modeling. Firstly, how can we divide $\mathcal{A}^\rho$ into $\mathcal{A}^{IS}$ and $\mathcal{A}^{IT}$ such that the total time of the application is minimal? Secondly, a joint problem is how can we determine the computational resource partitioning that, associated to the subdivision of $\mathcal{A}^\rho$, minimize the total makespan of the application? Those two questions are the core of the problem. Both questions are correlated between each other. To determine $\mathcal{A}^{IS}$ and $\mathcal{A}^{IT}$, it will be necessary to determine resource partitioning of the nodes and cores to be able to evaluate a subdivision of $\mathcal{A}^{\rho}$.

To tackle this issue, we propose to solve 1-ARP using two subproblems:

- **Sub-problem 1 (REPAS)**: This is a refinement of 1-ARP when the analysis sets are already determined. Given $\mathcal{A}^{IS}$ and $\mathcal{A}^{IT}$, determine a resource allocation of the compute units in between simulation and analysis tasks.
- **Sub-problem 2**: Using solutions for REPAS, divide $\mathcal{A}$ into $\mathcal{A}^{IS}$ and $\mathcal{A}^{IT}$ so that the total makespan of the application is minimized. We propose scheduling heuristics that test different possible task scheduling and their associated resource partitioning according to Sub-problem 1. All the possible schedules are not always suitable, due to the limited amount of memory of the system. Hence, we propose heuristics that respect system constraints in order to produce eligible schedules.

Then, in order to solve 1-ARP, we first study a refinement of it when the analysis sets are already determined. This is expressed as Problem 2.

Problem 2 (RESOURCE PARTITIONING FOR ANALYSIS SETS (REPAS))**.** *Given $\mathcal{A}^{IS}$, $\mathcal{A}^{IT}$ and $\mathcal{S}$, can we determine n^*, c^* and a scheduling of the N calls to the set $\mathcal{A}$ such that the total makespan of the application is minimal?*

This simpler problem REPAS provides a solution for 1-ARP when the scheduling of the analysis functions (either *in situ* or *in transit*) is given.

We show in the next section how to compute a solution to REPAS. We then use this solution in Section 4.4 to derive solutions to 1-ARP.

In this section, we study the REPAS problem. We first compute an optimal floating solution for the number of *in situ* nodes n^* and the number of helper cores c^*. We will then discuss an integer solution for this problem.

4.2 A solution to REPAS problem

We present in this section a solution for REPAS. We start by performing some rewriting to ease the resolution.

4.2.1 Reformulation of REPAS

We have seen in Section 3.3 of Chapter 3 that in the asynchronous case, the analysis from the iteration $i-1$ are performed at the same time as the simulation of iteration i. Hence the total execution time consists of the sum of: (i) the first iteration of the simulation; (ii) $\Pi - 1$ times the maximum time taken, either by the simulation, or by the analysis times; and (iii) the time for the final analysis.

Let us now denote by:

- $T_S(n^*, c^*)$, the time to execute one iteration of the simulation on n^* nodes, given that for each node, c^* cores are dedicated for in situ analysis;
- $T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*)$ the time to perform the analysis $\mathcal{A}^{IS}$ on n^* nodes, using c^* cores per node; and
- $T_A^{IT}(\mathcal{A}^{IT}, n^*)$ the time to perform the analysis $\mathcal{A}^{IT}$ given that the simulation is using n^* nodes.

Using notations above without their parameters to ease the reading, the execution time is:

$$T_S + (\Pi - 1)\max\left(T_S, T_A^{IS}, T_A^{IT}\right) + \max\left(T_A^{IS}, T_A^{IT}\right)$$

Assuming that Π is large enough, then

$$T_S + \max\left(T_A^{IS}, T_A^{IT}\right) << (\Pi - 1)\max\left(T_S, T_A^{IS}, T_A^{IT}\right)$$

and we can focus on the following optimization problem:

Problem 3. *Given $(\mathcal{A}^{IS}, \mathcal{A}^{IT})$, find $n^* \leq C_n$, and $c^* \leq c$ that minimize*

$$\max\left(T_S(n^*, c^*), T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*), T_A^{IT}(\mathcal{A}^{IT}, n^*)\right).$$

In the following, we show how to compute a solution to this problem, that is how to find n^* and c^*. We first write formally the different execution times.

Proposition 1 (Execution time of the different phases)**.** *Let X be the sum of the single core execution time of each $A_i \in \mathcal{A}^{IS}$ (i.e. $X = \sum_{A_i \in \mathcal{A}^{IS}} t_i(1)$). Similarly, we have: $\sum_{A_i \in \mathcal{A}^{IT}} t_i(1) = W - X$ where W is the total time of analysis tasks. Let Mem^{IT} be the sum of the memory peak of each $A_i \in \mathcal{A}^{IT}$ and b the bandwidth per node of the platform.*

$$\begin{aligned}
T_S(n^*, c^*) &= \frac{t_{sim}(1)}{n^* \cdot (c - c^*)} \\
T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*) &= \frac{X}{n^* c^*} \\
T_A^{IT}(\mathcal{A}^{IT}, n^*) &= \frac{W - X}{c(C_n - n^*)} + T_{Com}\left(n^*, C_n - n^*, \mathcal{A}^{IT}\right) \\
&= \frac{W - X}{c(C_n - n^*)} + \frac{Mem^{IT}}{(C_n - n^*) \cdot b}
\end{aligned}$$

These different costs come naturally from the fully parallel model, indeed it is the time to sequentially run all the tasks divided by the number of cores used.

4.2.2 Solution with rational number of cores and nodes

In this section, we compute a solution to Problem 3, under the form of rational numbers.

Theorem 1 (Optimal Number of Helper Cores)**.** *The optimal number of helper cores is given by the following:*

$$c^* = \frac{X \cdot c}{t_{sim}(1) + X}$$

Proof. To obtain this result, one can verify that in an optimal solution,

$$T_S(n^*, c^*) = T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*).$$

Indeed, otherwise if one is faster than the other one, we can share ε cores from the fastest computation to the slowest one such that its execution time does not increase too much its duration. This will reduce the slowest computation hence contradicting the optimality of the solution.

Hence we have:

$$\begin{aligned} \frac{t_{\text{sim}}(1)}{c - c^*} &= \frac{X}{c^*} \\ t_{\text{sim}}(1) \cdot c^* &= c \cdot X - c^* \cdot X \\ c^* &= \frac{X \cdot c}{t_{\text{sim}}(1) + X} \end{aligned}$$

□

Using the value of c^* from Theorem 1, we now compute n^*.

Theorem 2 (Optimal Number of in situ Nodes)**.** *The optimal number of* in situ *nodes is given by:*

$$n^* = \frac{X \cdot C_n}{c^* \cdot \left(\frac{W - X}{c} + \frac{Mem^{IT}}{b}\right) + X}$$

Proof. The result is obtained similarly to Theorem 1. First according to Theorem 1, we know that in the optimal solution the time for in situ analysis is equal to the simulation time. In addition, we use the formula from Problem 3 and verify that:

$$T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*) = T_A^{IT}(\mathcal{A}^{IT}, n^*),$$

where c^* is the value obtained in Equation (1) (and is a function of n^*). Similarly to the previous Theorem, one can verify that if one of the analysis is faster than the other one, we can share ε nodes from the fastest analysis such that its execution time does not increase too much. These nodes are then allocated to the other analysis which will reduce its execution time hence contradicting the optimality of the solution.

As we did before, we will solve the following equation in order to obtain the value of n^*:

$$\begin{aligned}\frac{X}{n^*c^*} &= \frac{W-X}{c(C_n-n^*)} + T_{Com}\left(n^*, C_n-n^*, \mathcal{A}^{IT}\right)\\ &= \frac{W-X}{c(C_n-n^*)} + \frac{Mem^{IT}}{(C_n-n^*)\cdot b}\end{aligned}$$

By rewriting,

$$\frac{X}{c^*} = \frac{n^*}{(C_n-n^*)}\cdot\left(\frac{W-X}{c} + \frac{Mem^{IT}}{b}\right)$$

Then,

$$n^*\cdot c^*\cdot\left(\frac{W-X}{c} + \frac{Mem^{IT}}{b}\right) = X\cdot(C_n-n^*)$$

This results in

$$n^*\cdot c^*\cdot\frac{W-X}{c} + n^*\cdot c^*\cdot\frac{Mem^{IT}}{b} + X\cdot n^* = X\cdot C_n$$

Finally,

$$n^* = \frac{X\cdot C_n}{c^*\cdot\left(\frac{W-X}{c}+\frac{Mem^{IT}}{b}\right)+X}$$

□

This result is derived for a linear bandwidth model but similar derivations can be performed with other communication models.

4.2.3 Integer solution for REPAS problem

In Section 4.2.2, we described a solution to compute the number of *in situ* nodes n^* and the number of helper cores c^*. However, these solutions return a rational number that is not suitable for us to describe a number of system physical resource.

To solve this issue, we round the result to the closest higher integer. Recall that in the proof of Lemma 1, the makespan of *in situ* analysis is computed to be at most equal to the simulation one, to avoid performance loss. Thus, we choose to round c^* and n^* to highest value to ensure that the highest makespan will be the simulation (in other words, analysis does not penalize simulation in the current setup).

This can lead to idle time for *in situ* resources. This will be the target of a future work to evaluate the best policy for c^* and n^* rounding.

In the following, we consider $\Pi = 1$ without loss of generality.

4.3 A solution to REPAS problem with Amdahl's law

In this section, we extend the results of Section 4.2 for a different task model: Amdahl's law [17].

4.3.1 Introduction to Amdahl's Law

In 1967, Gene Amdahl introduced a formula which gives the theoretical speedup in latency of the execution time of a task with fixed workload. Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. The principle is that a task submitted on a system can be divided into two parts:

1. a part that is sequential and thus, cannot take benefits from multiple resources

2. a parallel part that will improve when assigned to several compute units

In essence, this model states that a task will never scale perfectly when multiple resources are assigned to it due to inherent sequential nature of some part of the task.

Definition 1 presents a mathematical description of the law. For a task t with basic execution time T on a single resource and with a proportion of sequential code $0 \leq \alpha \leq 1$, the processing time of t using c resources is equal to the scaling over the c units of the $1 - \alpha$ original duration T, equals to $\frac{1-\alpha}{c} \times T$. In addition, there is the cost for the α proportion of sequential code whose time is αT.

Definition 1 (Amdahl's law)**.**

$$T(c) = \left(\alpha + \frac{1-\alpha}{c}\right) \cdot T \tag{4.1}$$

Some other task models such that Gustafson's law [68] can also be envisioned to enrich the model expressivity.

In this section, we present a new solution for REPAS problem where simulation follows Amdahl's law, and all analysis follows the basic fully parallel model. As simulation is a huge code, it is expected to be performed on an important set of nodes, hence having to face scaling problem. However, analysis are usually processed on few cores for which a fully parallel model is sufficient to model execution time. Even though some analysis may not scale correctly on staging nodes, we assume that the effect will be limited on overall performance.

4.3.2 Solution for resource partitioning subproblem

In this section, we first compute an optimal floating solution for the number of *in situ* nodes n^* and the number of helper cores c^*. We will then discuss an integer solution for this problem.

Remind that we want to solve Problem 3, as for perfectly parallel task model.

We first update the formal definition of the different execution times.

Proposition 2 (Execution time of the different phases)**.** *Let X be the sum of the single core execution time of each $A_i \in \mathcal{A}^{IS}$ (i.e. $X = \sum_{A_i \in \mathcal{A}^{IS}} t_i(1)$). Similarly, we have: $\sum_{A_i \in \mathcal{A}^{IT}} t_i(1) = W - X$ where W is the total time of analysis tasks. Let Mem^{IT} be the sum of the memory peak of each $A_i \in \mathcal{A}^{IT}$ and b the bandwidth per node of the platform. Finally, let α be the parameter of Amdahl's law, representing the fraction of the simulation that can be parallelized.*

$$\begin{aligned}
T_S(n^*, c^*) &= t_{sim} \cdot \left(\alpha + \frac{1-\alpha}{n^* \cdot (c - c^*)} \right) \\
T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*) &= \frac{X}{n^* c^*} \\
T_A^{IT}(\mathcal{A}^{IT}, n^*) &= \frac{W - X}{c(C_n - n^*)} + T_{Com}\left(n^*, C_n - n^*, \mathcal{A}^{IT}\right) \\
&= \frac{W - X}{c(C_n - n^*)} + \frac{Mem^{IT}}{(C_n - n^*) \cdot b}
\end{aligned}$$

These two analysis costs are the same as for the fully parallel model, indeed it is the time to sequentially run all the tasks divided by the number of cores used. Regarding simulation process, we use Amdahl's law [17] using n^* nodes with each $c - c^*$ cores at disposal.

Solution with rational number of cores and nodes

In this section, we compute a solution to Problem 3 when task follows Amdahl's law [17].

Theorem 3 (Optimal Number of Helper Cores)**.** *The optimal number of helper cores c^* and the optimal number of in situ nodes can be computed by solving a third order polynomial.*

Proof. As we previously mentioned, one can verify that in an optimal solution,

$$T_S(n^*, c^*) = T_A^{IS}(\mathcal{A}^{IS}, n^*, c^*).$$

Indeed, otherwise if one is faster than the other one, we can share ε cores from the fastest computation such that its execution time does not increase too much. This will reduce the slowest computation hence contradicting the optimality of the solution.

Hence we have:

$$t_{\text{sim}} \left(\alpha + \frac{1-\alpha}{n^*(c - c^*)} \right) = \frac{X}{n^* c^*} \tag{4.2}$$

$$t_{\text{sim}} \left(\alpha + \frac{1-\alpha}{n^*(c - c^*)} \right) = \frac{W - X}{c(C_n - n^*)} + \frac{Mem^{IT}}{(C_n - n^*) b} \tag{4.3}$$

We rewrite Equation (4.2) to express n^* as a function of c^*:

$$t_{\text{sim}}\alpha + \frac{t_{\text{sim}}(1-\alpha)}{n^*(c-c^*)} = \frac{X}{n^*c^*}$$
$$t_{\text{sim}}\alpha n^* + \frac{t_{\text{sim}}(1-\alpha)}{c-c^*} = \frac{X}{c^*}$$
$$n^* = \frac{X}{c^*\alpha t_{\text{sim}}} - \frac{1-\alpha}{(c-c^*)\,\alpha}$$

We now substitute the value of n^* in Equation (4.2):

$$t_{\text{sim}}\alpha = \frac{X}{\left(\frac{X}{c^*\alpha t_{\text{sim}}} - \frac{1-\alpha}{(c-c^*)\alpha}\right)c^*} - \frac{t_{\text{sim}}(1-\alpha)}{\left(\frac{X}{c^*\alpha t_{\text{sim}}} - \frac{1-\alpha}{(c-c^*)\alpha}\right)(c-c^*)}$$
$$t_{\text{sim}}\alpha = \frac{X}{\left(\frac{X}{\alpha t_{\text{sim}}} - \frac{c^*(1-\alpha)}{(c-c^*)\alpha}\right)} - \frac{t_{\text{sim}}(1-\alpha)}{\frac{Xc}{c^*\alpha t_{\text{sim}}} - \frac{X}{\alpha t_{\text{sim}}} - \frac{1-\alpha}{\alpha}}$$
$$\frac{1}{t_{\text{sim}}\alpha} = \frac{1}{\alpha t_{\text{sim}}} - \frac{c^*(1-\alpha)}{(c-c^*)\,\alpha X} - \frac{Xc}{c^*\alpha {t_{\text{sim}}}^2(1-\alpha)} - \frac{X}{\alpha {t_{\text{sim}}}^2(1-\alpha)} - \frac{1}{\alpha t_{\text{sim}}}$$

At the end, we obtain that

$$\frac{-c^*(1-\alpha)}{(c-c^*)\,\alpha X} - \frac{Xc}{c^*\alpha {t_{\text{sim}}}^2(1-\alpha)} + \zeta = 0 \tag{4.4}$$

where

$$\zeta = \frac{1}{\alpha t_{\text{sim}}} + \frac{X}{\alpha {t_{\text{sim}}}^2(1-\alpha)}$$

We multiply by $(c-c^*)$ in Equation (4.4) to get

$$\frac{-cc^* + cc^*\alpha + c^{*2} - c^{*2}\alpha}{\alpha X} + \frac{c^*cX - c^2X}{c^*\alpha {t_{\text{sim}}}^2(1-\alpha)} + \zeta(c-c^*) = 0$$
$$\leftrightarrow c^{*3}\cdot a + c^{*2}\cdot b + c^*\cdot c + d = 0 \tag{4.5}$$

□

with

$$a = \frac{1-\alpha}{\alpha X}$$
$$b = \frac{c\alpha - c}{\alpha X1} - \zeta$$
$$c = \frac{cX}{\alpha {t_{\text{sim}}}^2(1-\alpha)} + \zeta c$$
$$d = \frac{c^2X}{\alpha {t_{\text{sim}}}^2(1-\alpha)}$$

Solving this third order polynomial gives a solution for c^*. With this solution, one can compute the value of n^* using Equation (4.2) or (4.3).

This section shows that our approach can be extended to other task models. In practice, due to the rounding of the solution, both solutions are expected to return very similar results. Future work will be dedicated to the comparison of the model performance with different task models. In the remaining of this first part of the manuscript, we consider the perfectly parallel task model with the associated solutions we above derived for its simplicity in this modeling approach.

We now move one step forward in the resolution of the global optimization problem by proposing scheduling heuristics of the different tasks of the application.

4.4 Sub-problem 2: task scheduling decisions

With the solutions to Problem 3, we now want to solve the general 1-APPLICATION RESOURCE PARTITIONING problem. Indeed, we are now able to determine a resource partitioning given a division of the analysis tasks between *in situ* and *in transit* processing. However, all the possible schedules are not always suitable, due to the limited amount of memory of the system. As we discussed in Sections 3.2.3, let us define procedure *VIABILITY* that, for a given system, verifies that the *in situ* nodes have enough memory for simulation and the memory peaks associated to the scheduled *in situ* analysis. As we previously stated, the set of *in transit* analysis do not require memory to be performed (cf. Section 3.2.3). However, they generate a communication time for sending their input to the *in transit* nodes. We now propose different scheduling heuristics for simulation and analysis tasks.

4.4.1 "One-by-One" greedy algorithms

A natural polynomial strategy for scheduling algorithm is to greedily move analysis functions from $\mathcal{A}^{IT}$ to $\mathcal{A}^{IS}$, and to compute the optimal allocation of nodes and cores. Note that because of memory constraints, not all analysis can fit *in situ*, which is why we consider that they all start as *in transit* analysis.

Such an algorithm is described by Algorithm 1. The idea is to sort the analysis functions following a given metric (priority order). We initialize the algorithm with $\mathcal{A}^{IT} = \mathcal{A}$ and $\mathcal{A}^{IS} = \emptyset$. Then we greedily move each analysis according to the priority order one by one from $\mathcal{A}^{IT}$ to $\mathcal{A}^{IS}$. When an analysis is moved, we determine the minimum number of nodes needed so that the required memory for simulation plus the memory reserved for *in situ* analysis is not greater than the memory available. If the number of nodes is greater than C_n, then we leave this application in $\mathcal{A}^{IT}$. Otherwise we compute the optimal allocation of nodes and cores by a procedure called *PTNG*, that applies Lemma 3. Procedure 2, described by Algorithm 2, ensures the respect of memory constraints.

In this work we try the following priority functions:

Algorithm 1 *Generic Greedy Algorithm*

Require: Set of analysis Ana, total number of nodes C_n, total number of cores c, memory per node m, bandwidth per node b, simulation time $\mathcal{S}$, memory consumption of simulation m^S, a metric $metric$ to sort the analysis

Ensure: A resource partitioning n^*, c^* and a scheduling of tasks Ana^{IS}, Ana^{IT}

$Ana^{IS} \leftarrow \emptyset$
$Ana^{IT} \leftarrow Ana$
$Analysis \leftarrow metric(Ana)$
$(n^*, c^*) \leftarrow \mathit{PTNG}\left(Ana^{IS}, Ana^{IT}, C_n, c, b\right)$
exec_time $\leftarrow \mathit{SCHED}\left(C_n, n^*, c, c^*, \mathcal{S}, Ana^{IS}, Ana^{IT}, b\right)$
for $a_i \in Analysis$ **do**
    $Ana^{IS} \leftarrow Ana^{IS} \cup a_i$
    $Ana^{IT} \leftarrow Ana^{IT} \setminus a_i$
    $(n_1, c_1) \leftarrow \mathit{PTNG}\left(Ana^{IS}, Ana^{IT}, C_n, c, b\right)$
    if $\mathit{VIABILITY}\left(Ana^{IS}, n_1, m, m^S\right)$ **then**
        e $\leftarrow \mathit{SCHED}\left(C_n, n_1, c, c_1, \mathcal{S}, Ana^{IS}, Ana^{IT}, b\right)$
        if e $<$ exec_time **then**
            $n^* \leftarrow n_1$
            $c^* \leftarrow c_1$
        else
            $Ana^{IS} \leftarrow Ana^{IS} \setminus a_i$
            $Ana^{IT} \leftarrow Ana^{IS} \cup a_i$
        end if
    else
        $Ana^{IS} \leftarrow Ana^{IS} \setminus a_i$
        $Ana^{IT} \leftarrow Ana^{IS} \cup a_i$
    end if
end for
return $\left(Ana^{IS}, Ana^{IT}, n^*, c^*\right)$

- INCREASING-TIME/DECREASING-TIME: we sort the analysis by their increasing/decreasing execution time on a single core.
- INCREASING-PEAK/DECREASING-PEAK: we sort the analysis by their increasing/decreasing memory peak.
- RANDOM: we compute a uniformly random priority order.

DECREASING-TIME is well known to be efficient for scheduling applications with an objective of minimizing the execution time. INCREASING-TIME is famously known to be efficient with respect to the objective of improving fairness. RANDOM is here as a witness: anything worse than this heuristic can be considered as a poor order.

Algorithm 2 *Viability of a resource partitioning for a set of* in situ *analysis*

Require: Set of *in situ* analysis Ana^{IS}, number of *in situ* nodes n^*, memory per node m, memory consumption of simulation m^S

Ensure: $True$ if the *in situ* input system would be viable, $False$ otherwise

$result = False$
$IS\_memory \leftarrow (m \times n^*) - m^S$
if $(IS\_memory \geq 0)$ **then**
  $counter \leftarrow 0$
  for $ana \in Ana^{IS}$ **do**
    $counter \leftarrow counter + (memory\_peak\,(ana))$
  end for
  if $counter \leq IS\_memory$ **then**
    $result \leftarrow True$
  end if
end if
return $result$

4.4.2 Optimal scheduling algorithm

To compare the previous algorithms, we will use the optimal scheduling algorithm that tests all the possible *in transit*/*in situ* configurations and keep the one that generates the lowest execution time. This optimal algorithm is described by Algorithm 3.

4.4.3 Complexity analysis

The greedy algorithms have a linear complexity on the number of analysis tasks $\mathcal{O}(K \log(K))$. *VIABILITY* procedures have a complexity in $\mathcal{O}(K)$ and *PTNG* in $\mathcal{O}(1)$.

The optimal algorithm has a complexity exponential on the number of analysis functions $\mathcal{O}(2^K)$. Indeed, it has to generate all possible subsets of K functions. As usually there is a small number of analysis functions, this exponential factor remains limited (cf. Table 1 in [46] for an example).

This concludes this chapter on proposing automatic tools to perform resource partitioning and task allocation for *in situ* processing of applications. Next chapter is dedicated to the evaluation of the different heuristics as so as the general performance of the models we proposed.

Algorithm 3 *Optimal Algorithm*

Require: Set of analysis Ana, total number of nodes C_n, total number of cores c memory per node m, bandwidth per node b, simulation time $\mathcal{S}$, memory consumption of simulation m^S
Ensure: A resource partitioning n^*, c^* and a scheduling of tasks Ana^{IS}, Ana^{IT}

```
$Ana^{IS} \leftarrow \emptyset$
$Ana^{IT} \leftarrow Ana$
subsets ← generate_subsets $(Ana)$
exec_time ← SCHED $\left(C_n, n^*, c, c^*, \mathcal{S}, Ana^{IS}, Ana^{IT}, b\right)$
$(n^*, c^*) \leftarrow$ PTNG $\left(Ana^{IS}, Ana^{IT}, C_n, c, b\right)$
for set in subsets do
    $IS \leftarrow$ set
    $IT \leftarrow$ ana \ set
    $(n, HC) \leftarrow$ PTNG $(IS, IT, C_n, c, b)$
    if VIABILITY $\left(IS, n, m, m^S\right)$ then
        e ← SCHED $\left(C_n, n, c, HC, \mathcal{S}, Ana^{IS}, Ana^{IT}, b\right)$
        if e < exec_time then
            $n^* \leftarrow$ n
            $c^* \leftarrow$ c
            $Ana^{IS} \leftarrow IS$
            $Ana^{IT} \leftarrow IT$
        end if
    end if
end for
return $\left(Ana^{IS}, Ana^{IT}, n^*, c^*\right)$
```

Chapter 5

Evaluating Model Through Simulation Process

Contents

In this chapter, we present an evaluation of our model and scheduling strategies on synthetic applications. Section 5.1 discusses the simulation setup and methodology. Section 5.2 evaluates our proposed solutions for asynchronous *in situ* processing. In Section 5.3, we discuss from these results the important features that determines the application performance. Finally, Section 5.4 studies the performance on the synchronous scenario.

5.1 Evaluation methodology

To evaluate the different strategies, we designed a simulator to study the performance of the different scheduling algorithms (Section 4.2 in Chapter 4) coupled with our resource partitioning solution (Section 4.4 in Chapter 4). Each scheduling heuristic of Section 4.4 is implemented and returns an execution time for a given simulation function, set of analysis and platform. The platform is given as a number of nodes, cores, a total amount of memory M and a bandwidth. We recall that every node has the same amount of memory, and bandwidth.

Recall that we consider here a perfectly parallel task model. A simulation task is given as a processing time on single core and a peak memory consumption. We fix as a constant the memory consumption of the simulation to be 1 and the total available memory of the system to be 1.33. This is justified by the fact that usually, simulation is a large task that requires most of the system resources. These values can be easily tuned in the setup of the simulator to satisfy a wide range of scenarios.

The analysis functions are described by the same features as the simulation and are randomly generated regarding the following process. We fix a desired memory occupation M for the application (simulation and analysis) and a number of analysis functions K. Then, we randomly generate a set of K analysis for which the total amount of memory peaks sums to $M - 1$. The execution time on one core for each function is randomly picked between an upper bound and lower bound percentage of the simulation execution time. In the following, we fix K to 8. Here again, each parameter can be tuned by users.

The simulator is designed to support both asynchronous analysis scenario and synchronous (in this case the *in situ* analysis is part of the simulation). For the latter scenario, we use the same scheduling and model tools previously mentioned. The simulation memory peak and work are updated with the scheduled *in situ* analysis.

Given this simulator, we perform different evaluations of the model and algorithms by increasing the memory load of the application and studying the impact of the memory constraint on the analysis execution mode. We vary the memory load of the application from 1.05 to 2.[1] To ensure that the communication time does not interfere in the results, we tested different bandwidth per node values going from 10% of the total memory per unit time to 100%.

To ensure the reliability of results, we perform 130 samplings for each memory occupation and plot the average of the results for each algorithm. The number of nodes is 50 and number of cores per node is 8.

The simulator has been developed using SageMath[2]. The plots of this section are generated using R language. The code of the simulator and all details related to software dependencies[3], plot generation or installation instructions are detailed in Appendix A.1.

5.2 Results in asynchronous scenario

In this section, we present the results of the above setup in the asynchronous scenario.

Figure 5.1 presents the performance of algorithms where each node has a bandwidth equals to 10% of its memory per unit time. The first plot presents the execution time of the application with regards to memory occupation. The execution times are normalized with regard to the optimal Algorithm 3.

First of all, we note that the algorithms tend to converge to the optimal when the constraints of memory are either strong or weak. This is explained by the fact that if

[1] When the memory load equals to 2, the memory consumption of analysis and simulation are equal. In reality, this extreme point is never reached.

[2] http://www.sagemath.org/

[3] The simulator has not been tested on other versions of SageMath than 8.1 but there should not have any problem as we use Sage as a runtime, we do not use its provided libraries.

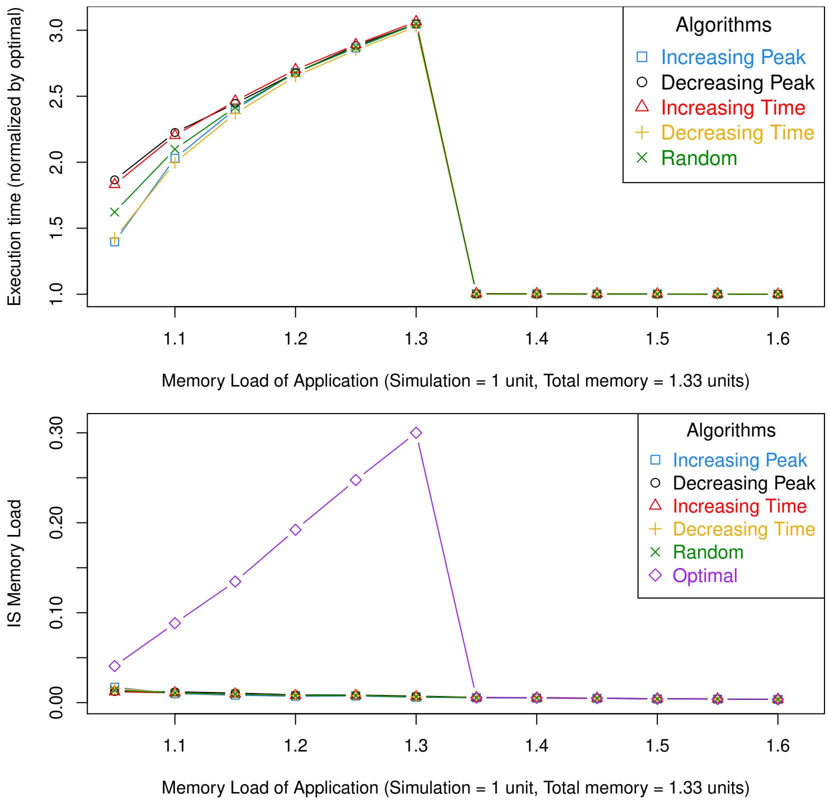


Figure 5.1: Simulation of Algorithm performance for asynchronous scenario with bandwidth is equal to 10% of the memory per node, per unit time.

the memory load of application overpasses by far the total memory of the platform, the only solution is to offload all the analysis *in transit* to maintain simulation performance (we assume that the simulation can fit on the platform by using all its resources). In the contrary, if the memory constraints are very low, the optimal solution is to perform *in situ* analysis by sharing the resources between analysis and simulation. However, greedy heuristics are not able to find an appropriate *in situ* analytics schedule due to the fact that they consider analysis one by one. Figures 5.2, 5.3 and 5.4 show that different bandwidths do not influence the scheduling policy.

From the performance analysis, we extract two algorithms that seems to perform better than others: INCREASING-PEAK and DECREASING-TIME. They also induce an *in situ* memory occupation relatively close to the optimal. To understand part of their good performance, we have also plotted the *in situ* memory occupation for each algorithm with regards to the total memory occupation of the application. The observation that one can make is that the performance of algorithms seems correlated to the memory size occupied by *in situ* applications. It seems to make sense as one may expect that the additional cost incurred is the one due to data movement, hence one wants to minimize

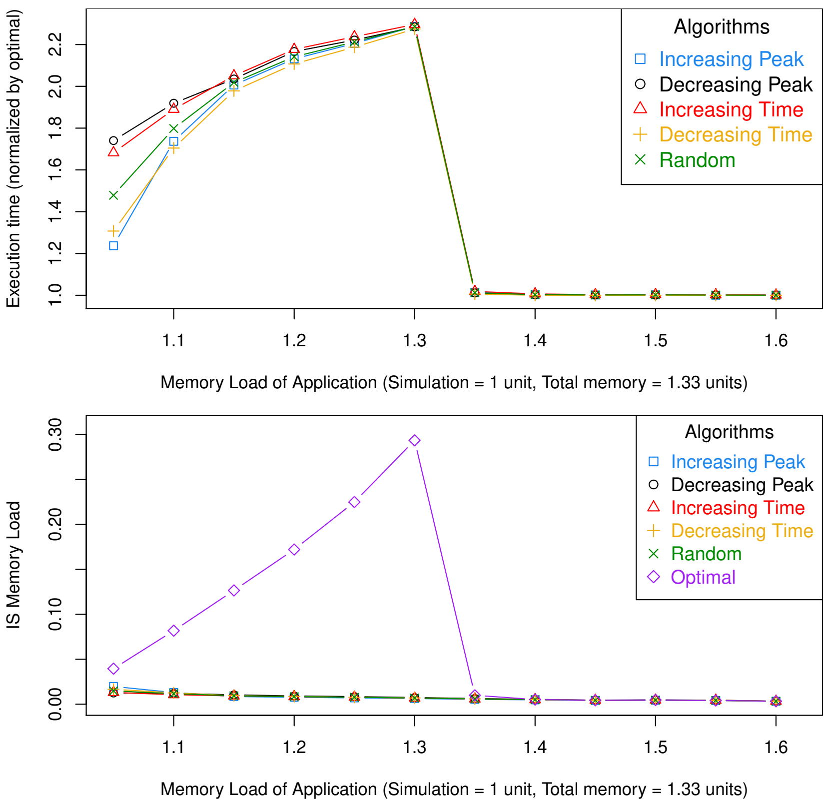


Figure 5.2: Simulation of Algorithm performance for asynchronous scenario with bandwidth equals to 25% of the memory per node, per unit time.

the amount of such movement.

However, an interesting conclusion is that our greedy algorithms are often performing badly with regards to this optimal. One of the reason is surely that those algorithms consider analysis one by one, and not by packs. It seems intuitive that scheduling a group of tasks on a given set of resources has more probability to induce better performance in an overall perspective than only one task. In the future, we must take into account batches of analysis rather than one by one in order to design efficient scheduling policies.

5.3 Criteria for application performance

From previous discussion on asynchronous scenario, one can deduce that an important feature for system performance is resource utilization. If some resources are reserved for *in situ* analysis, those resources must be sufficiently used in terms of memory and computation to improve system performance. Otherwise, those resources are better be

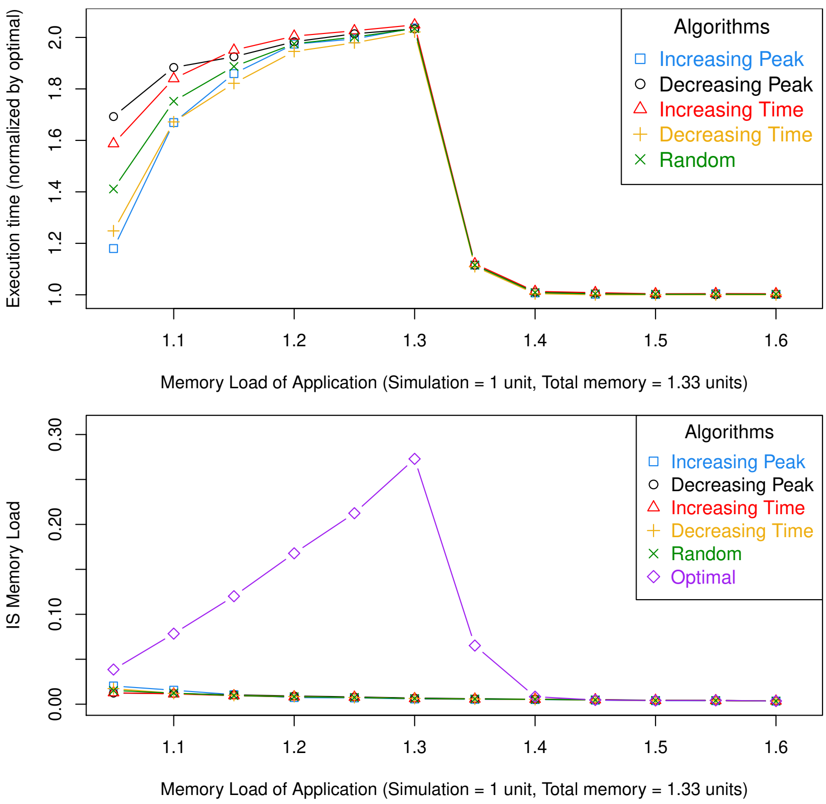


Figure 5.3: Simulation of Algorithm performance for asynchronous scenario with bandwidth is equal to 50% of the memory per node, per unit time.

left to simulation and the analysis offloaded into *in transit* processing.

It is not a trivial result as it seems. Indeed, if an analysis is fast but uses lots of memory, it may mean that the helper cores are never used. One may expect that it would be better to use them. Note that this trade-off may explain the good performance of DECREASING-TIME. While INCREASING-PEAK is better at using as much memory as possible for analysis, DECREASING-TIME might be better at using the helper cores as much as possible.

To validate this hypothesis, we plotted Figure 5.5. The x-axis shows the *in situ* memory load of considered algorithms (normalized by optimal algorithm). The farther a point is on the right, the better the scheduling algorithm uses the *in situ* memory. The y-axis is dedicated to (normalized) execution time: the lower a point is, the better the associated algorithm performs. We included in this plot all points of Figure 5.1 to Figure 5.4. We shape each point by its algorithm and color it with a gradient representing its normalized *in situ* workload.

One should read the figure as follows. In a first time, we look where most of the points are located regarding x-axis. Indeed, if a non negligible number of points are

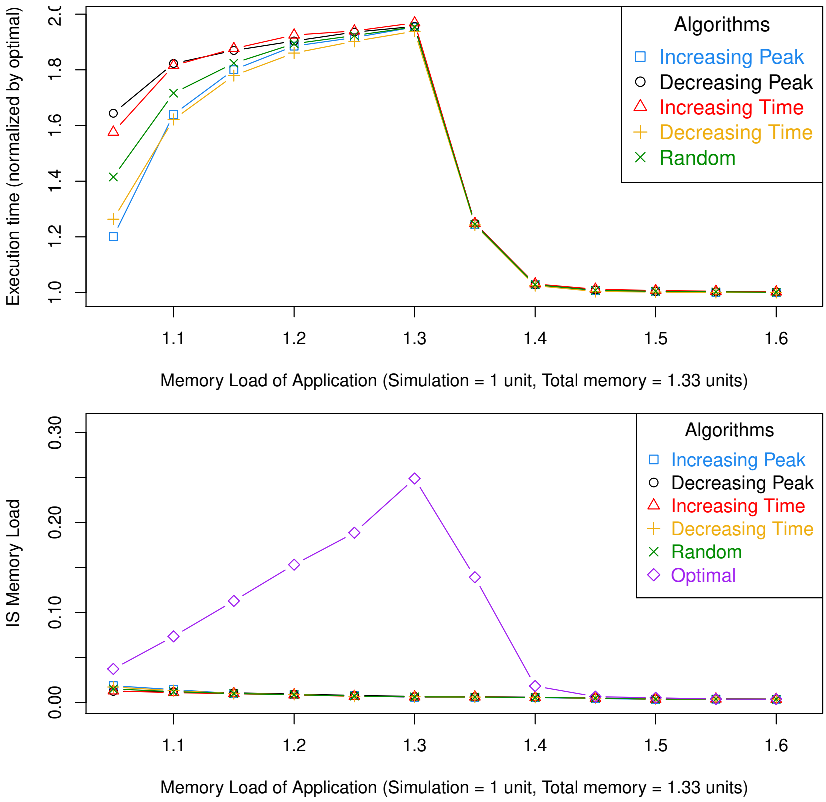


Figure 5.4: Simulation of Algorithm performance for asynchronous scenario with bandwidth is equal to 100% of the memory per node, per unit time.

greater than 1 on this axis, it means that the optimal solution does not have a strong correlation with *in situ* memory usage. We clearly see that most of the points have a value less than 1. The exception points stand for cases with lowest application memory load. Hence, the optimal solution is strongly correlated to the *in situ* memory usage. This confirms the previous intuition. Let us now take a look on the y-axis and the coloring corresponding to *in situ* workload. We clearly see a trend between the performance of the algorithm and the *in situ* workload. Indeed, the best performance are obtained when the *in situ* workload is closed to optimal, hence indicating a correlation in optimal solution between performance and *in situ* workload. All features together, this plot shows the strong correlation between the *in situ* resource usage and algorithm performance.

Finally, Figure 5.6 shows the evolution of the three different makespans (simulation and analysis) of optimal solution for a bandwidth at 10% of memory per node. An interesting remark is that the optimal solution is often a mixture of *in situ*/*in transit*. We see that the simulation is always the larger makespan until the memory load of application overcomes system capacity. This comes from the rounding strategy discussed in Section 4.2.3. Once the memory load of the application reaches the system memory

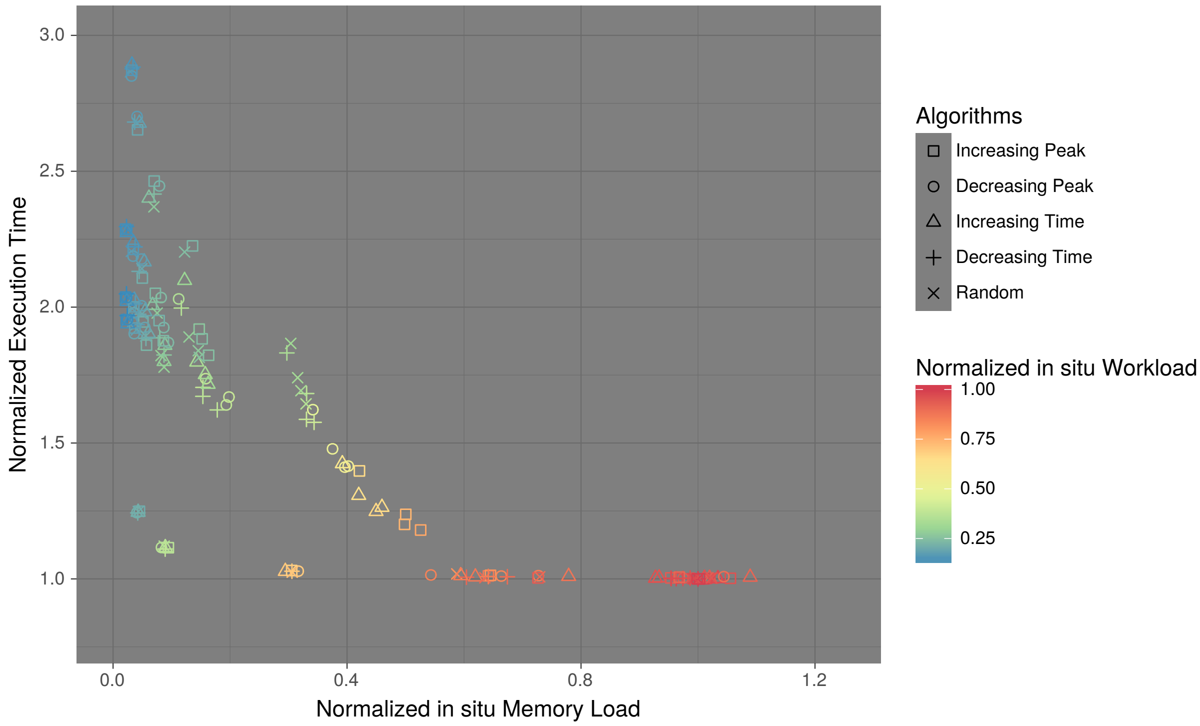


Figure 5.5: Evaluation of *in situ* memory and workload parameters in algorithm performance for asynchronous scenario.

capacity, the analysis have to be offloaded *in transit*, thus generating a communication overhead as we can see in the figure.

Figures 5.7-5.8 show different trends for INCREASING-PEAK and RANDOM heuristics[4]. Due to their greedy behavior, those heuristics generate a schedule with heavy load of *in transit* analytics, hence important transfer overhead. Thus, the *in transit* analytics makespan is the main cost of each iteration for greedy heuristics, explaining the gap of performance with optimal solution.

5.4 Results for synchronous scenario

In this section, we compare the performance of asynchronous versus synchronous analytics. We expect the synchronous scenario to outperform the asynchronous one. Recall that when synchronous, we iterate the simulation over all *in situ* cores, then we pause it to perform the *in situ* analysis on the same cores as the *in transit* analytics on its dedicated nodes. In the latter scenario, the working surface dedicated to simulation and analysis is more important than the one in the asynchronous case, thus induces better performance. In practice, synchronous analytics causes complex side effects such as

[4]The trends are similar for other heuristics and different bandwidths.

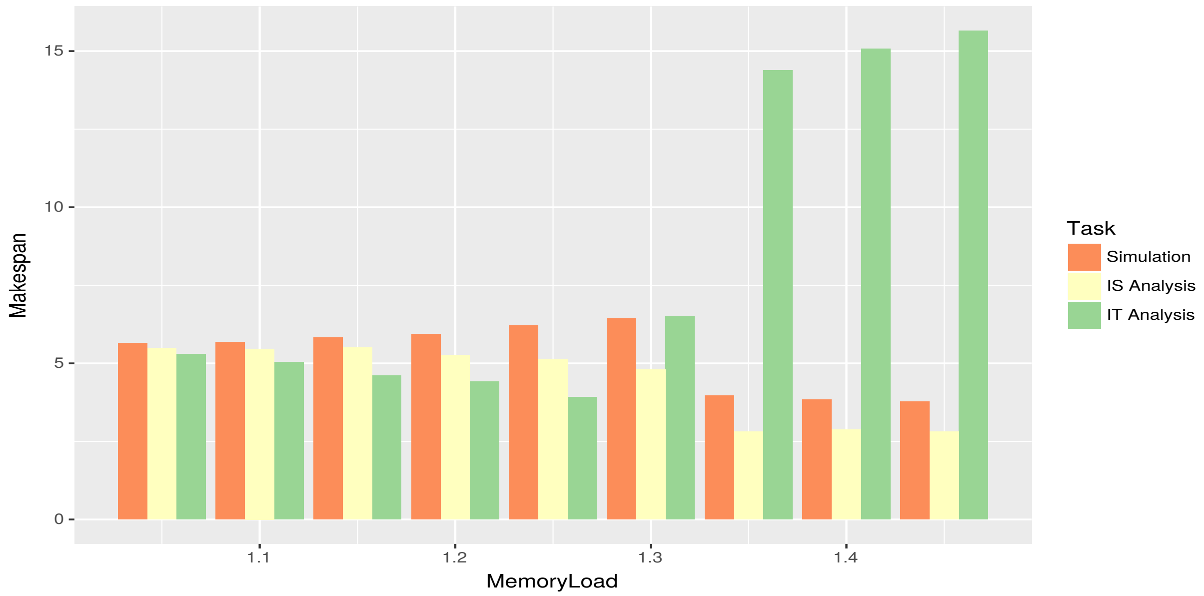


Figure 5.6: Comparison of simulation, *in situ* and *in transit* analysis makespans for different application memory load for Optimal solution when bandwidth per node equals to 10% of memory per node.

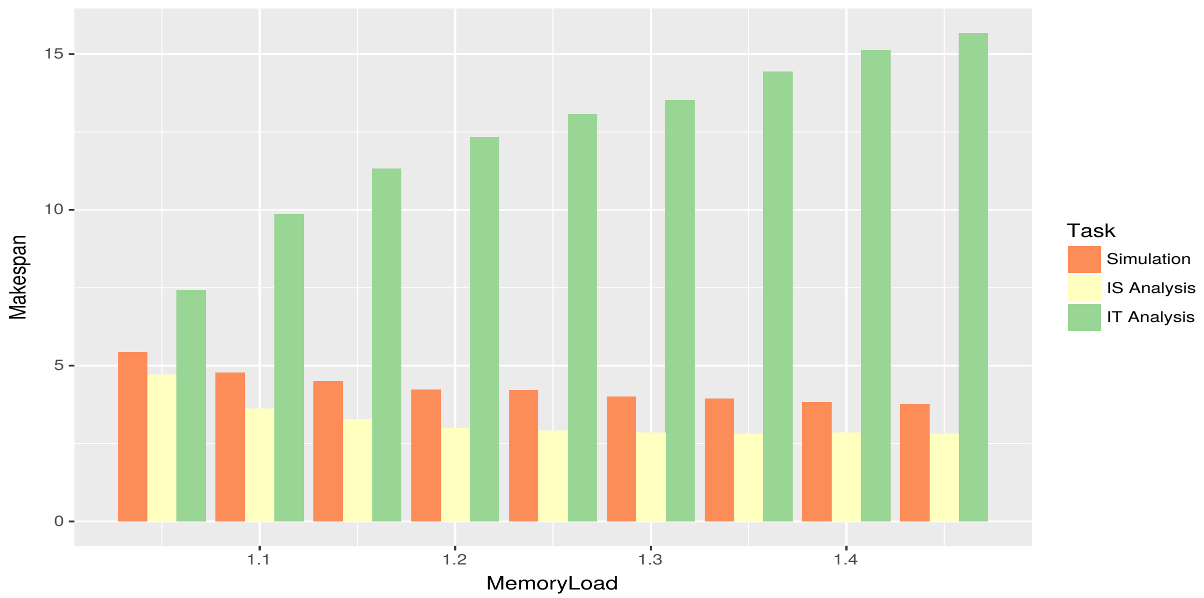


Figure 5.7: Comparison of simulation, *in situ* and *in transit* analysis makespans for different application memory load for INCREASING-PEAK heuristic when bandwidth per node equals to 10% of memory per node in asynchronous scenario.

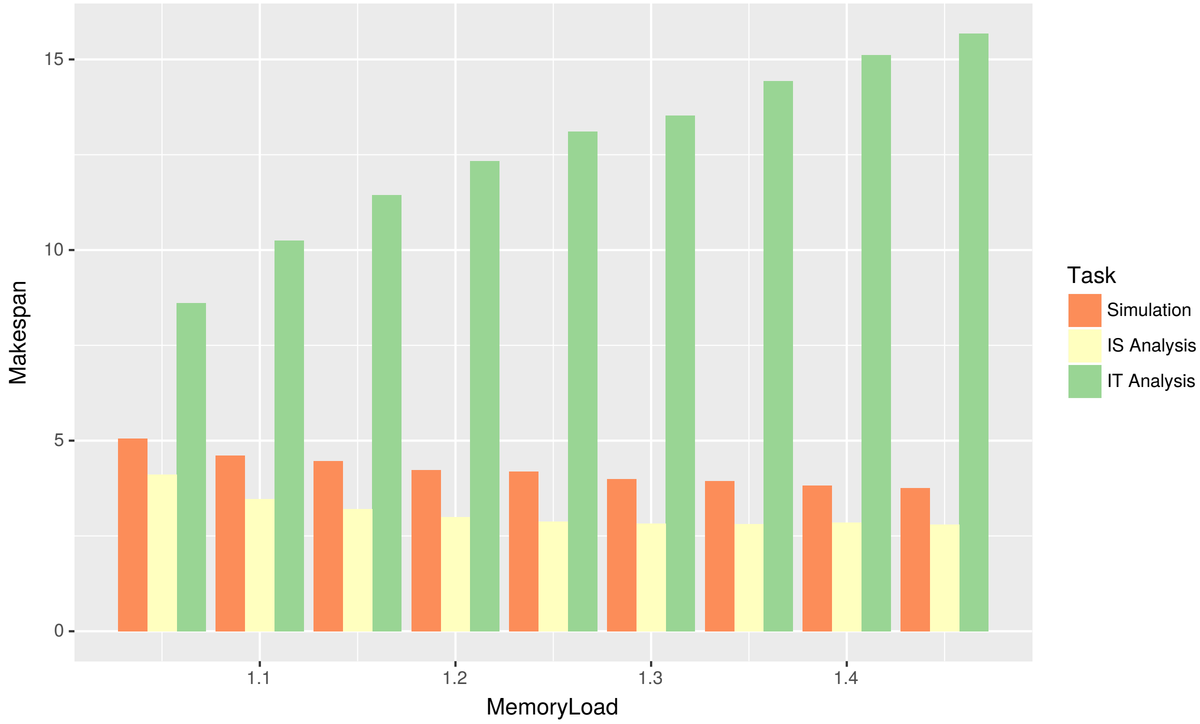


Figure 5.8: Comparison of simulation, *in situ* and *in transit* analysis makespans for different application memory load for the RANDOM heuristic when bandwidth per node equals to 10% of memory per node.

cache pollution overhead when switching between tasks and it is intrusive to the code. Moreover, simulation is often not able to efficiently scale while we increase the number of cores. A task model such as Amdahl's law [17] would even reinforce this effect. From this perspective, asynchronous analytics is more beneficial despite the data copies it engenders. This would be the target of a future work. For now, the synchronous scenario represents a lower bound on application performance as we consider embarrassingly parallel tasks.

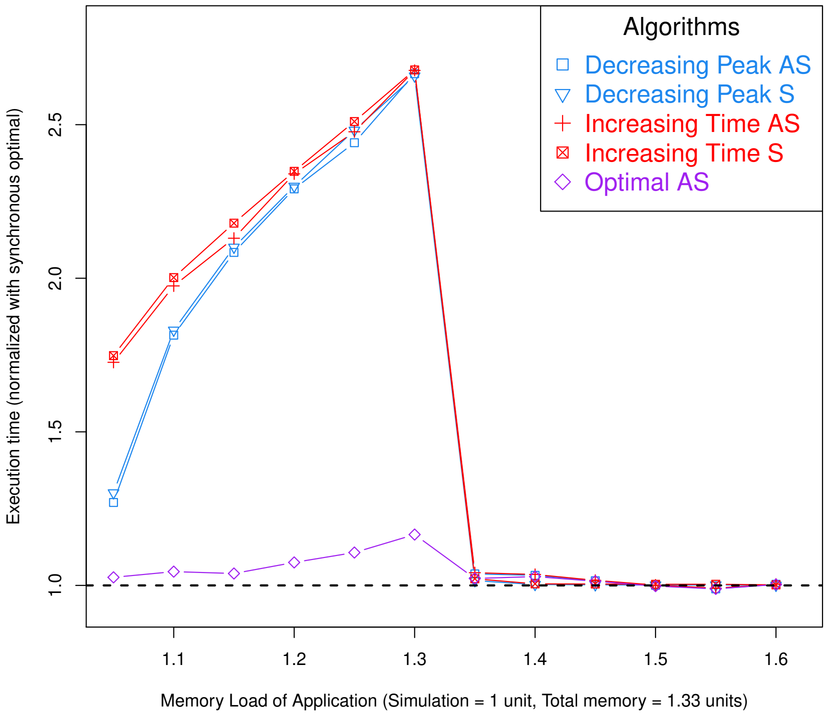


Figure 5.9: Simulation of Algorithm performance in synchronous scheme with bandwidth is equal to 25% of the memory per node, per unit time.

Figures 5.9-5.10 show the simulation results in the synchronous case for a setup similar to the one in Section 5.2. We plot in those figures the two best algorithms (DECREASING-PEAK and INCREASING-TIME[5]) and also the optimal solution of asynchronous scenario, all normalized by the optimal synchronous solution. We see that two optimal solutions have at most a 10% difference and tend to be very similar when the application memory load overflows system capacity. We observe for the greedy

[5]The order of performance for all other heuristics in synchronous mode is the same as in the asynchronous results.

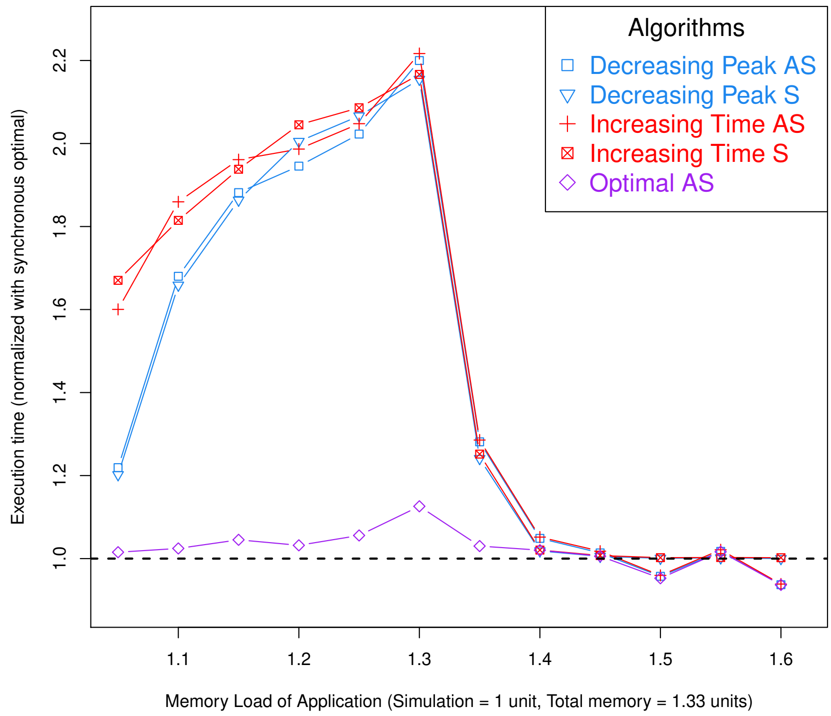


Figure 5.10: Simulation of Algorithm performance in synchronous scheme with bandwidth is equal to 100% of the memory per node, per unit time.

heuristics that the synchronous mode does not help to enhance performance. Some difference may occur in some cases, but never more than a 10% gap between the performance of a heuristic in synchronous/asynchronous processing. With a larger number of cores per node, we know that this difference will tend to reduce, due to the flexibility the increasing number of cores provides.

We see the benefits of synchronous execution mode for *in situ* processing when we have *in situ* analytics to process. However, when the application memory load rises, we see that the optimal synchronous can be outperformed by the optimal asynchronous. This is due to the fact that, in synchronous, there is not enough *in situ* work to take the benefits of all *in situ* resources when the simulation is paused. In contrast, asynchronous scenario induces a more flexible resource partitioning with less *in situ* resource loss.

This concludes the performance evaluation of our models and algorithms.

Chapter 6

Summary of Part I

In this chapter, we summarize the contributions presented in this first part of the manuscript. We also introduce ideas of short-term future works.

6.1 Summary

In this first part of the manuscript, we considered the problem of optimizing data-intensive applications on supercomputers. Coming from scientific fields such as biology or physics, these applications are composed of two phases: the simulation, being compute-intensive, and the analytics, presenting a data-intensive profile. Such applications exhibit a regular execution time due to the structure of their code, developed as monolithic blocks. Hence, determining the duration of the resource reservation can be accurately performed after some basic profiling of the two phases.

As variations in the performance of I/O operations are one of the major bottleneck in supercomputers, the storage of all the simulation data on disks prior to analytics process is no more sustainable. As an alternative, coupling simulation and analytics in order to post-process data directly on their production site has emerged as a major trend for data-intensive applications. We considered in our work the widely used *in situ* paradigm where simulation and analytics are iteratively run on shared or non-shared resources.

Critical issues in *in situ* paradigm is to make decisions about the compute nodes and workload distribution in between simulation and analytics. This problem is two-fold: it consists in partitioning correctly the resources shared between the simulation and the analysis functions (nodes, memory, etc.), and scheduling the different analysis in order to perform them *in situ* or *in transit*.

As a solution to these issues, we proposed a general model of the workflow of these applications and formulated an optimization problem for minimizing the execution time of the applications. We designed automatic tools to ease the resource partitioning and computational load distribution between the application and the machine resources. We derived a model to partition the compute nodes between the different set of tasks of the application. This model relies on several assumptions that we have either tried to keep as inconsequential as possible, or that are easily modifiable (e.g. bandwidth model, task model) in our analysis. We also provided scheduling heuristics to decide which analysis functions have to be performed *in situ* and *in transit*. We established the flexibility of our models by extending our solutions to another task model for the simulation, Amdahl's law. We assumed that analytics is composed of a limited number of analysis functions, according to usual implementations of analytics. Indeed, some use cases from computational physics involves limited number of analysis functions, according to Table I in [46]. In general, some routines are in charge of performing visualization duties, while others process verifications on data integrity and some manage the data saving on disks. Some other functions can be scheduled to perform post-processing of data (extraction, statistics etc), depending on the problem under study. Our model is flexible in terms of number of analysis functions, as well as it can be adapted for many different frameworks.

Finally, we performed an evaluation of our solutions on synthetic applications. We were able to assert that the memory usage of the analysis functions is one of the key feature to account for when performing the scheduling of analysis functions. Specifically, when partitioning analysis functions between *in situ* and *in transit*, one needs to maximize the amount of analysis functions computed *in situ*.

6.2 Perspectives

Near-future work will be dedicated to evaluating experimentally these results. Simulation results must be validated in the framework of a real simulation process with its own analytics workflow. This evaluation would allow us to study how robust the assumptions made by our model are, and if our algorithms can be used as such, or if we need to make our model more precise. To do so, we plan to use the FLowVR middleware[1] [53]. FlowVR enables to launch parallel simulations on thousand of cores with the *in situ* paradigm. FlowVR is based on a data-flow approach, where the application is modeled as a set of components that exchange messages, that is, data. Hence, the application is expressed as a directed graph. FlowVR automates the deployment of the application on machine resources, and the communications between the different components using daemons running on each distributed resource. Overall, it offers a flexible environment to design customized analytics associated to a simulation code. In our case, the simulation is performed through the Gromacs package that emulates molecular dynamics with equations of motions considering until millions of particles. The associated analytics are designed to perform operations on and tracking of the particles. For instance, some analytics are dedicated to perform visualization of the system under study. Some others are designed to compute metrics that evaluate the evolution of the system. These

[1] `http://flowvr.sourceforge.net/`

experimentations are currently being designed on the Occigen[2] machine of the french National Computing Center for Higher Education.

Once this first evaluation will be over, a natural next step is to focus on designing algorithms that maximize the usage of *in situ* resources (memory and cores). A first idea would be to consider the analysis by packs rather than one by one. In general, we would like to quantify the bound that makes *in situ* analysis interesting from a performance perspective.

Several other directions include enriching our model to enhance its expressivity for different types of simulation and analytics coupling. Extensions to heterogeneous nodes is a first step, with different processing speed or specialization in certain type of operations. Indeed, it can be interesting for applications to perform computations on different types of CPU depending on the tasks that compose its workload. For instance, we introduced that AI techniques such that Machine Learning functions are now being used on HPC facilities. Such AI frameworks usually better perform on GPU than CPU. In the second part of this manuscript, we study such type of AI applications and propose solutions to efficiently manage them on HPC facilities.

Another direction on optimizing data-intensive applications is to better take into account the hierarchical memory of CPU. As we previously stated, different levels of memory are available for compute cores to store data. With their different access speed and capacities, it is necessary to optimize the usage of this hierarchy in order to ensure performance of the running tasks. Such improvement would imply a good knowledge of the memory usage of the different tasks of the application. This can be done, for example, by annotating data to help users determine their memory space destination. Understanding the I/O profile of the applications can also be useful to perform the scheduling of applications.

Finally, different communication models can be considered in order to better reflect the competition for I/O system and more accurately predict the transfer overhead for the different data.

[2]Machine configuration is available at https://www.cines.fr/en/supercomputing-2/hardwares/the-supercomputer-occigen/configuration/.

Part II

On the Use of Stochastic Applications on HPC facilities

Chapter 7

Introduction

Contents

7.1 Introduction

In the first part of this thesis, we mentioned that high performance computing platforms rank amongst the most powerful structures able to perform heavy-load critical computations. We described typical HPC applications as sets of massively parallel codes that require a significant number of computing resources to meet their need for memory and computation. Fields such as astronomy and cosmology, computational chemistry, particle physics, and climate science have evolved together with the advance of platform architecture and software stack. These scientific applications follow widely used programming models in order to reach massive levels of parallel processing. The classical *fork-join* paradigm consists in branching off the execution of the program into several parallel tasks at pre-defined moments of the application execution. Branches are later merged and joined to go back to initial sequential processing. This branching principle is useful for increasing the amount of memory available for the application because each branch gets processed on a dedicated resource. This workflow model is actually widely used for developing scientific applications, which are composed of a set of tasks that have dependencies between each other. Most of the time, such applications can be represented by a directed acyclic graph where nodes stand for the different tasks of the application, and the edges are the dependencies between them. The advantage of such application representations is that they provide users with a better control over the flow of data generated by the different tasks. In this paradigm, each task comprises either several sequential tasks, or a large set of parallel tasks, for instance using MPI. Many workflow system management have been designed in recent years in order to offer users solutions to develop their applications in many different environments, whether it be a local cluster, a grid (a network of compute nodes spatially distributed) or large-scale infrastructures. The Pegasus project [44] is an example of such workflow management system. It has been designed to offer portability, scalability, performance, but also reliability by being able to detect errors occurring during application lifetime.

Originating from BigData, some similar programming frameworks have appeared on HPC environments, owing to the convergence between the two domains. One of the most famous one is the *MapReduce* model. It has been developed with the motive of processing parallelizable problems across large sets of input data on many-node infrastructures. Due to the constant increase of input datasets, these frameworks are now targeting the large-scale infrastructures of HPC, since local clusters cannot offer that much computational power. One of the strength of MapReduce lies in its ability to take advantage of the locality of data. Indeed, it is able to process the data near the place where they are stored in order to avoid unnecessary communications. Input are divided in between each processing unit, called worker. Each worker first applies a *Map* function on its local input, and saves the generated output on a temporary storage. A shuffling operation is then executed in order to reassign the data based on output keys generated by the *Map* phase. This process ensures that the data that have the same key get located on the same worker. Finally, each worker simultaneously processes in parallel each data, which have been grouped by key. In short, both *Map* and *Reduce* operations can be performed in parallel by each node.

Newly emerging applications move beyond a structure with large monolithic codes that are usually designed for HPC, which use tightly-coupled and compute-centric algorithms. Fields such as neuroscience, bioinformatics, genome research, computational biology usually carry out exploratory research that involves more dynamic, heterogeneous, multi-phase workflows using ad-hoc computations and methodologies. New Machine Learning (ML) and AI frameworks have become important tools in exploratory domains. While significant efforts have been made over past years to improve these ML techniques, research advances have induced new requirements in terms of computations. For instance, Deep Learning requires an important training part, in which the quality of the model is supposed to increase as the dataset size grows.

Such workflows involving ML techniques will soon start targeting HPC infrastructures that offer high computation support as well as substantial memory and good network specifications. However, the profiles of these emerging applications differ from that of classic HPC applications. Often, the execution time of these applications is difficult to estimate because they are input-independent. It is common for such applications to have walltimes ranging between several hours and several days. This feature constitutes a real limitation for users for whom requesting the maximum possible walltime often induces an overestimation that increases the total cost of the request. For second generation applications, this paradigm works differently. Users still need to estimate the peak memory requirement, so as to mobilize only the amount of resources that will meet peak memory needs. However, important variations in memory consumption can have a significant impact on performance prediction. Studies using machine learning methods to estimate the future resource consumption of an application assume a constant peak memory footprint (e.g. [134]). However, it remains tricky to evaluate the exact walltime of the application. Correspondingly, it is hard to estimate the reservation duration. If one overestimates the walltime of the application, the cost for the user can prove to be very important. By way of contrast, underestimation can lead to application failure. A solution to this problem would be to continue processing a sequence of reservations until the application terminates. We propose to tackle this problem from a theoretical perspective in order to determine efficient scheduling heuristics to minimize the cost of processing a stochastic application.

In the remaining part of this introductory chapter of the second part of this thesis, we will study the profile of stochastic applications with the goal of understanding the properties and characteristics of these new frameworks. To that end, we will focus on a representative neuroscience application named SLANT [80, 79]. This code features the typical behavior of the upcoming stochastic applications, that is to say:

1. Its workflow consists of multiple stages and a walltime comprised between tens of minutes to hours depending on the hidden characteristics of the input Magnetic Resonance Imaging (MRI) and of the hardware of the platform;

2. While its peak memory requirement is predictable, within one execution the memory footprint can have variations of tens of GBs;

3. Its code is dynamic, in continuous development, and dependant upon on the needs of each study.

7.2 Emergence of a second generation of applications

In this section, we present the high-level observations of a representative stochastic application used in the neuroscience field, called SLANT (presented in Section 7.2.1). From these observations, we will propose in Chapter 9 a generic application model that we will use in Chapter 10 to design scheduling strategies for such types of applications.

7.2.1 Spatially Localized Atlas Network Tiles (SLANT)

Spatially Localized Atlas Network Tiles (SLANT) [80, 79] is a Neuroscience application developed by the medical and neuroscience department at the Vanderbilt University [91]. Most of the work presented in this second part of the manuscript has been done in collaboration with Vanderbilt University. This cooperation was quite profitable since we could have access to emerging applications and explore them so as to provide solutions to help developers execute efficiently their applications.

SLANT features an easily understood input data, MRI images, which makes it ideal to be studied, and it is actually representative of many of this new type of HPC applications. For example the RADICAL-Pilot job system to develop bioinformatics workflows is often used to create workflows that spawn large numbers of short-running processes that can exhibit highly irregular I/O and computation patterns [109]. Similarly, applications using Adaptive Mesh Refinement (AMR) methods have been proven to have highly unpredictable performance variations depending on the characteristics of the input data [142].

In our case, SLANT performs multiple independent 3D Fully Convolutional Network (FCN) for high-resolution whole brain segmentation. It takes as input a MRI image obtained by measuring spin–lattice relaxation times of tissues[1].

We run the application on a Haswell platform hosted on Plafrim[2], using the Singularity container runtime. Due to the hardware limitations of the Haswell platform[3], we could not run the GPU version of SLANT. Instead, we used a CPU version of the application whose code is freely available at `https://github.com/MASILab/SLANTbrainSeg`. The CPU version is given in the form of a Docker image. Since the Haswell platform does not support Docker either, we used a software called Singularity, which allows us to utilize the Docker image without installing it on the platform. Appendix A.3 presents the details of our protocol to run the application while Appendix D.1 describes the architecture of the Haswell platform.

We are interested in the behavior of the applications when it does not get impeded by any interference coming from the system or from some other applications (e.g. congestion due to shared resources).

Different versions of SLANT exist, depending on whether the network tiles are overlapped or not. Here, we consider the overlapped version (SLANT-27 [80]), in which the target space is covered by $3 \times 3 \times 3 = 27$ 3D FCN. The application is divided into three main phases: i) a preprocessing phase that performs transformations on the target image (MRI is a non-scaled imaging technique) ii) a deep-learning phase iii) a post-processing phase doing label fusion so as to generate the final application results. Each task may present run-to-run variation in their walltime.

7.2.2 High-level observations

We previously mentioned that second generation applications present variation in their walltime. In this section, we will attempt to verify this assertion, and investigate the origin of this variation at application level. To this end, we will run the SLANT-27

[1] The format of the input MRI is named T1weighted image.
[2] `https://www.plafrim.fr/`
[3] The GPU version requires *nvidia-docker* program that is not available on the Haswell platform.

application on different input types. These inputs are extracted from OASIS-3 [90][4] and *Dartmouth Raiders Dataset* (DRD)[5] [73] datasets.

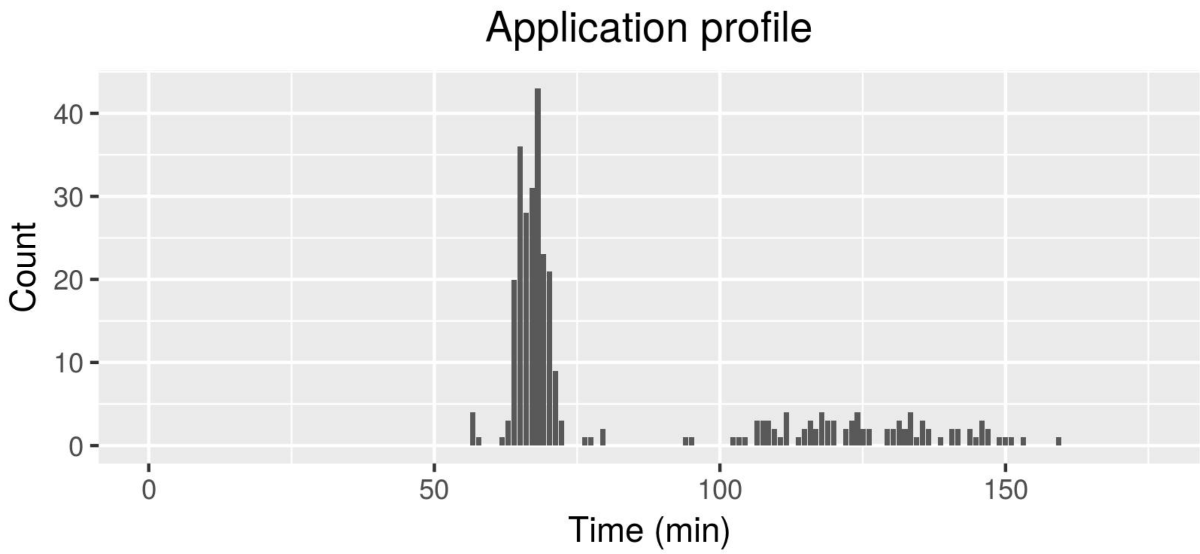


Figure 7.1: SLANT application walltime variation for various inputs.

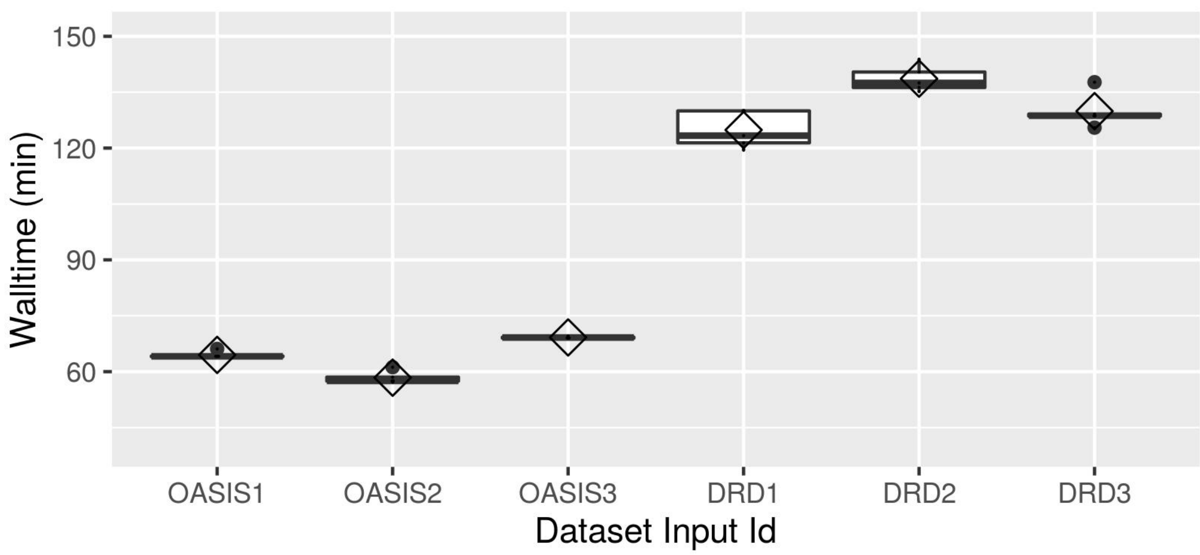


Figure 7.2: Performance variability on identical inputs. Variability is studied over five runs.

[4]For this very large dataset, we only used a subset of available data. Selected data are presented in Appendix A.3.
[5]Available at http://datasets-dev.datalad.org/?dir=/labs/haxby/raiders

In Figure 7.1, we confirm our previous observations about the large walltime variations. What is more, we can distinguish two categories of walltimes, which correspond to the two datasets: OASIS inputs have a walltime of $70\text{min}\pm15\%$, and DRD inputs have a walltime of $125\text{min}\pm30\%$. The obvious questions arising from these observations are the following:

- Is the walltime variation due to a machine artifact (or is it due to the *quality* of the input)?
- Is the walltime variation due to the input size (and can it be predicted using this information)?

We study these questions hereafter in the present section via a set of experiments. First, we randomly selected three inputs of both datasets, and executed them five times each. We present the results in Figure 7.2. We can see that the behavior for each input is quite consistent. There are slight variations for DRD inputs, but nothing of the same magnitude as the one that was observed over all inputs. Hence, it seems that the duration of the execution is highly influenced by the input.

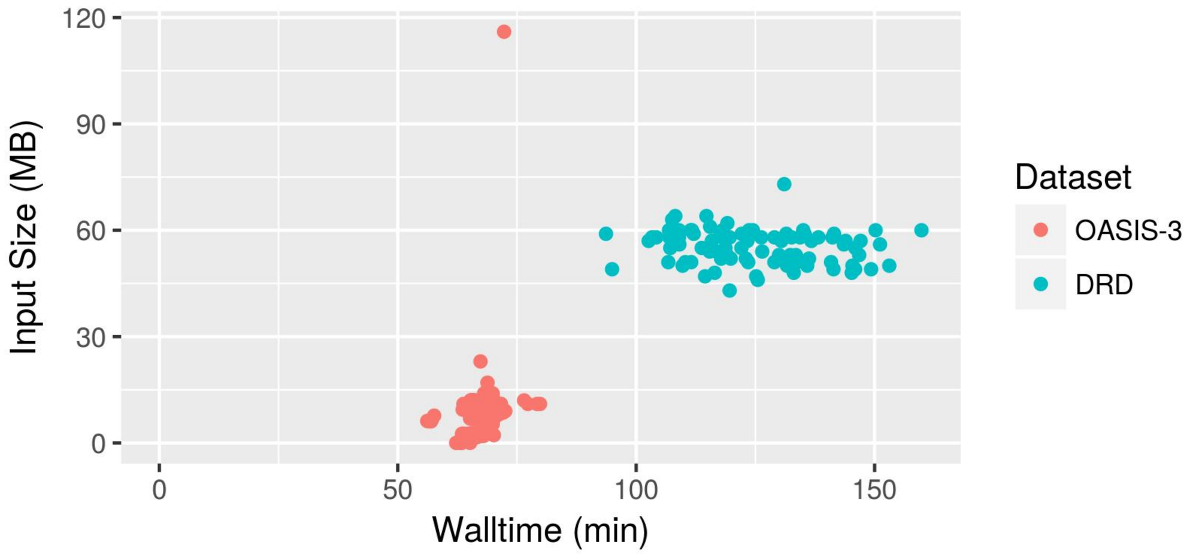
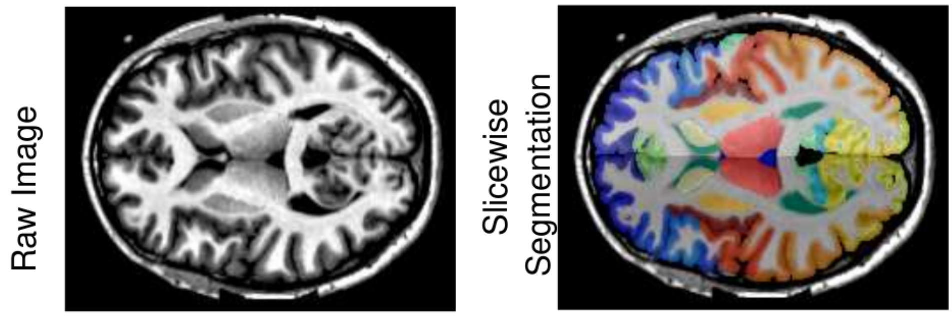
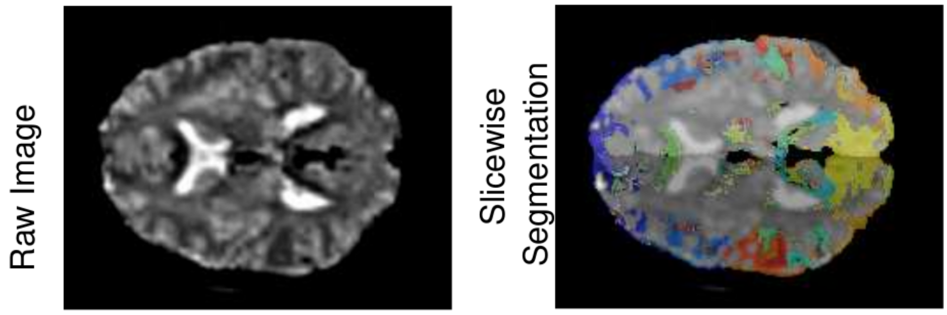


Figure 7.3: Correlation between the size of the input and the walltime over the 312 runs.

We then study the variation of walltime as a function of the input size in Figure 7.3. We can see that for a given dataset, the walltime does not seem correlated with the input size. The corresponding Pearson correlation factors are 0.30 (OASIS) and -0.15 (DRD). The datasets, however, appear to have different input types: except for the outlier at 120 MB, the input sizes of OASIS vary from 0 to 30MB while those from DRD vary from 45 to 75MB. We present visually the type of inputs for the two databases in Figure 7.4. Intuitively, the performance difference between DRD and OASIS is probably due to the resolution of the images.



(a) Segmentation for OASIS.



(b) Segmentation for DRD.

Figure 7.4: Typical inputs and outputs based on the dataset.

Altogether, we can make the following preliminary observations about these new applications:

1. We observe large variations in terms of execution time of Neuroscience applications, complicating their execution on HPC platforms. This aspect will constitute the core of the remaining research hereafter exposed in this part of the manuscript

2. These variations are mostly determined by elements from the input, but are not correlated with the size of the input (*quality* and not quantity).

At this stage, one can argue that our analysis does not take into account machine-related performance variations (I/O, shared resources etc), nor does it try to see if one could "predict" the performance of the application based on inputs. yet these omissions were purposeful. Many work in the scientific literature focus on both these topics, for example [164, 51, 92] for machine-related variations and [134] for predicting application makespan based on inputs (see Chapter 8 for more details). Here, we specifically aim at showing application specific variations which we believe is a new way of handling HPC applications (as shown in Figure 7.2, we verified that, for given inputs, there was almost no machine-related variations). Obviously, one needs to ultimately take into account many aspects when taking scheduling decisions but, from a research standpoint, we believe that it is important to study them separately. In this work, we focus on the variability that are due to the application itself.

7.3 Contributions

The second part of this manuscript focuses on the study of the type of emerging applications that we shall call *Stochastic Applications* in the remaining part of this manuscript.

In Chapter 8, we review a few related work dealing with scheduling and managing such applications on reservation-based platforms.

Chapter 9 will be devoted to the application model, platform cost model and problem statement. Based on our observations about SLANT, we will propose a generic application model in which the execution time of the application follows a known probability distribution. Some assumptions are made to tackle the problems, such as a flat memory model for the applications. Our study aims to bridge the gap between the specific characteristics of exploratory applications and the strict requirements of HPC batch schedulers that hinder productivity and innovation for new computational methods. Using this application model to formulate different cost models of platforms, we will then describe different optimization problems of scheduling stochastic applications on a reservation-based platform. The objective is to minimize the expected cost of a successful execution of the jobs. Our cost model of platforms will basically cover two main scenarios: a model that considers the possibility of checkpointing in the end of a reservation (that is, the fact of saving the progress of application to avoid losing it in the end of the reservation), or a model that does not comprise any checkpointing. This method will allow us to cover a wide range of target applications and frameworks.

In Chapter 10, we will detail a few algorithmic solutions to the problems stated in Chapter 9. We will derive our solutions from the cases in which application walltime follows either continuous or discrete probability distributions. In short, our suggestions are based on two different scenarios (checkpointing is available or not). Depending on the type of distribution, the solutions are either exact solutions, near-optimal approximation algorithms, or sub-optimal ones.

Afterwards, we present in Chapter 11 an extensive set of simulations and experiments with the aim of evaluating the scheduling strategies that we will previously have introduced. We will perform evaluations for a wide range of usual probability distributions, with many different cost functions covering a wide range of scenarios. We will show that our solutions outperform current approaches for stochastic application in HPC systems.

In Chapter 12, we will try to move one step forward in this study by attempting a more in-depth analysis of the SLANT application. We will propose an analysis at task level of the profile of the application in order to extract important features of the stochastic nature of the application. Our goal is to validate and enhance our first application model. One of the main innovations in this part is that we will not consider anymore the memory model to be flat. We will propose a decomposition of the application into sub-tasks having their proper memory peak. Then, based on these new observations, we will put forward new scheduling heuristics inspired by the ones previously presented in Chapter 10. To this end, we will use these models to estimate the resource request for SLANT when deployed on a HPC system. We will then evaluate these new strategies alongside the ones described in Chapter 10. Eventually, we will point out that a good knowledge of the application is quite crucial to the design cost-efficient scheduling strategies.

Finally, Chapter 13 propose a conclusion on the second part of this manuscript.

Chapter 8

Related Work

Contents

In this chapter, we explore some related work about scheduling stochastic application and checkpointing.

8.1 Reservation-based scheduling

In this section, we briefly recall the principle of HPC schedulers and discuss the importance of resource estimation for their performance. More details about it can be found in Chapter 2 of Part I.

8.1.1 HPC schedulers and resource estimation

Submitting a job onto HPC platform consists in sending a request to a complex program called scheduler. Each request details the amount of resources needed (number of nodes/cores as well as optionally the type of nodes and/or the amount of memory per core required by the application). A request must also provide a total duration for which resource will be allocated. Then the scheduler takes all this information into account when mapping the different user requests with the machine resources. Most schedulers for HPC systems (Slurm [155], Torque [131] or Moab [37]) implement an iterative algorithm triggered by state changes, such as new job submissions, job starting or ending event, or timeout. This principle works well for scientific applications since the amount of resources needed can be estimated prior to application execution with a good precision. We see that this is more complex with stochastic applications, for which the walltime for a given input cannot be estimated accurately.

Resource overestimation during submission is a typical strategies for HPC applications since the cost of getting your application killed due to underestimation is very high. As it was recently empirically showed by our colleagues Gainaru et al. [60], the runtime overestimation due to the inherent structure of stochastic jobs can impact both system utilization and user response time by 25-30%. These observations have motivated the collaboration with Vanderbilt University on the design of scheduling strategies for stochastic applications.

Several works aim at improving the use of batch scheduler in the presence of uncertainty on the runtimes. Zrigui et al. [165] discussed using online learning to improve the performance of batch schedulers by a simple classification of jobs into two categories, small and large. BigData frameworks such as MapReduce [42] and Dryad [84] rely on schedulers (e.g. YARN [143] and Mesos [77]) with distinct features such as fairness, or resource negotiation to manage the workload. However, accurate application needs must be known to the scheduler. The strategies we propose aim at providing hints to the user so they can optimize their submissions, but also to these communities since their schedulers may use user-given execution-time distributions of tasks to implement their own sequence of reservation with checkpointing.

To provide solutions in the presence on uncertain execution time, some work focus on optimizing the expected response time of applications by performing distribution fitting [34, 87, 64, 111, 115]. They assume a well-known probability distribution of the job execution time. These ideas were extended to provided near-optimal reservation strategies in both HPC and cloud systems for a set of stochastic jobs with backfilling [60]. These papers do not consider a task model for the stochasticity of the application because they simply focused on the execution time (flat memory model), as we do in Chapter 10 and 11. The problems under study are already complex to tackle, and some assumptions need to be done in order to develop a solution. However, we move one step further in Chapter 12 by developing a stochastic task model, which allows to develop a memory footprint model. Hence, we obtain a better representation of the memory behavior of the applications.

8.1.2 Pricing and reservation schemes in the cloud.

Cloud computing platforms have emerged as another option for executing HPC applications. Job scheduling in the cloud has an even bigger challenge [67], since it needs to deal with highly heterogeneous resources with a wide range of processor configurations, interconnects, virtualization environments, etc.

Different pricing and reservation schemes are also available for users who submit jobs to a cloud service. The former usually provides a more flexible service while the latter incurs a much cheaper cost. The user pays for the resource requested and/or used based on an hourly or daily rate. Several works have been done to study these schemes in the cloud, and from a computer science perspective. Many of these studies focus on the pricing strategies and service management of platform providers [151, 45, 39, 14, 130]. Some works consider modeling the delays for users [14], and how providers manage the idle resources [45]. The work in [151] studies the pricing practices of Amazon AWS [16] when the price is dynamically adapted to real-time demand and idle resources. In [39], authors provide an analytical model of pricing for reservation-based scheme (used by Amazon AWS) and utilization-based scheme (used by Google GCP [65]). They show that the effective price mainly depends on the variation of platform usage and the competition for customers. Some tools are also provided for users to perform cost evaluation in order to select which type of platform to use. They show that users with high-volatility demand should consider using AWS offers while one should use GCP in the other case. Our experimental results in Chapter 11, compared with on-demand or utilization-based services, reservation strategies can provide cost-effective options for executing stochastic jobs when there is significant difference in the offered price.

8.2 Scheduling under variability

In this section, we study related work of scheduling under variability at application or system level.

8.2.1 Variability at application level

Scheduling with uncertainty Many prior works have considered stochastic scheduling for jobs with execution time uncertainty. Most research in this paradigm (e.g., [34, 87, 64, 111, 121, 115]) assume that the execution time of a job follows a known probability distribution and aims at optimizing either the expected response time or the makespan for a set of jobs under various distributions. Most of them, however, do not consider the problem in the context of reservation-based scheduling. In this manuscript, we also model jobs by an execution time following a probability distribution. From this, we propose reservation strategies for a single job in both HPC and cloud systems, either with or without checkpointing considerations.

More specifically, many works deal with stochastic job scheduling (e.g., [137, 49, 140, 36, 147]). Various models [35] have been proposed to model the performance of executing stochastic jobs on computing platforms. For instance, in [95], stochastic jobs are modeled as a DAG of tasks whose execution times and communication times are stochastically independent. The authors in [137] propose a model based on resource load in grid systems. Several refinements can be envisioned, such that improving scheduler performances by including distribution features in order to optimize final performance. Also, dealing with heterogeneous nodes increases problem complexity [140]. A book by Pinedo [122] contains a comprehensive survey of stochastic scheduling problems, and a book chapter [62] proposes a detailed comparison of stochastic task-resource systems.

Prediction of execution time While the second part of this document focuses on working with the uncertainty of execution time, another complementary direction is to try to remove this uncertainty by predicting the execution time.

The predictive methods based on machine learning, rely on supervised inductive learning over historical log files on large-scale compute clusters, using either predicted memory usage of the jobs or predicted the execution time of the jobs, and assume a large set of training data. As instance, in [134], the authors use five types of regression algorithms on a large dataset (millions of entries) containing past executions of applications on their internal cluster, and predict both the memory and the processing time of future runs. The study in [18] combines CPU and GPU execution historic logs and generate observations that help users or administrators to classify jobs into equivalence classes by likelihood of failure. In [89], the authors use a predictive scheme for identifying small walltime jobs. With a similar approach, Gaussier et al. [63] introduced several machine learning methods for predicting the class of execution (small/large) for HPC application with the goal of improving scheduling and backfilling algorithms. A similar regression model is presented in [66], but with a focus on predicting underestimation of job runtime for applications running on TSUBAME 2.5. Closer to our study, [107] focuses on two bioinformatics applications. Their method is capable of increasing the accuracy of predicting the job execution time, the memory needs, and the space usage. However, the method requires a large training set. Unlike these studies, our applications are extremely dynamic with their codes in continuous change. Thus they require a strategy that not only is capable of dealing with stochasticity in memory and execution time, but can learn the behavioral pattern of the application fast. Hence, the huge amount of data required to train the prediction model is a limitation for our considered applications. Moreover, we discussed in Figure 7.3 the difficulty of predicting application walltime with regards to input size.

8.2.2 Variability at system level

In our work, we focus more on application-level variability. However, variability at system level is also a factor to take into account when one studies the variability of applications in terms of walltime or memory needs.

Several works [13, 88] study the factors of variability in HPC machines at (among others) system level. In [128], authors point out the importance of cross-application contention and system activity. Some works [83] also study the variability due to component manufacture.

Other system constraints such as I/O interference [153, 100] or including consideration of network traffic [51], power limits [83] or concurrency tuning in the HPC middleware [118], can also become a significant reason for performance variability. Interferences are also observed between different virtual machines running on the same hardware in a cloud computing provider [123]. Although we could include all these variability causes in our study, we chose to focus on application-specific variations, a new trend in HPC, and separate their impact from the hardware constraints.

Finally, variation in resource requirements is a known fact for HPC even for existing traditional applications. It can be attributed to several factors: randomized algorithms, inherent job variability (e.g. depending on input data), resource sharing, interferences, OS jitter, etc. Inherent job variability is the topic of this work, and includes iterative methods that work towards convergence [142] through discrete steps or studies that trigger an in-depth analysis of subproblems based on certain observations. Those will experience variability in both execution time and memory consumption. It has also been recently observed in machine learning framework on GPUs [97].

8.3 On the use of checkpointing

In this section, we present the current approach for checkpointing in HPC, and present the current solutions for stochastic applications.

Checkpointing is a major tool to cope with stochastic applications and/or platform unavailability. It is usually implemented through checkpoint-restart [150, 82]: saving a snapshot of application state on a persistent external support (typically, the central filesystem). Then, this snapshot of the application can be used to resume computation due to either failure of the application itself or hardware/platform crash.

The objective is to perform this process with minimal loss of computing work. Hence, one has to balance the gain of restarting from a snapshot after a failure with the time spent to checkpoint. Checkpointing too often is indeed not an efficient solution because saving huge amounts of data usually slows down the application significantly. In the context of fault-tolerance, a lot of work (e.g., [157, 41, 82]) has been devoted to deriving the optimal checkpointing interval that minimizes the checkpointing overhead or resource waste. A checkpoint can be taken using a variety of techniques at every level of the system, from utilizing special hardware/architectural checkpointing features through modification of the user source code. Some works [55] present the implementation of checkpoint-restart in HPC system.

For this work, we consider more a user perspective for stochastic application. This is justified by the possibility to save only the necessary data, which could lead to save checkpoints smaller than a whole system snapshot. This has the double advantage of reducing the required storage space, as so as limiting the use of I/O systems whose performance is affected by contention [153]. Also, user-level checkpoints could be used for many different purposes such as fault-tolerance or debugging purpose.

Specifically, for application-level checkpointing, we could consider checkpointing performed either by the application itself explicitly modifying the code to work with a user level checkpoint library (like FTI [27]) or by linking an external library. In our work, we focus on this latter case because it generally does not require to modify the application, which we do not aim at doing now. BLCR [70] was a popular solution but it does not seem to be maintained anymore, and does not support containers as far as we know. DMTCP [19] is a more recent alternative that has good support for parallel HPC jobs, and may be integrated with Slurm [125]. However it lacks container supports.

Another solution is to use CRIU [1], which is well supported in the upstream Linux kernel, and has support for checkpointing containers [110, 120]. CRIU enables checkpointing a running process under the form of a collection of files(open files, open sockets etc), later used for restoration point from where the snapshot has been taken. At restore process, CRIU reads the files in the checkpoint, forks the process then restores its resources (memory, sockets, timers, etc.). However, the application ends after CRIU has performed the checkpoint, which can be a limitation. Also, Docker container support [40] seems still experimental. Thus, checkpointing in practice is a complex process to handle for stochastic applications, especially when using a container runtime. Some surveys [126] discuss the advantages and drawbacks of each approach above described.

In our experiments in Chapter 12, we use CRIU to checkpoint the docker image of SLANT. Our work is actually not strongly tied with CRIU. Hence this choice may be revise in the future if target applications require MPI support or do not need containers.

In this work, we present strategies that combine reservation and checkpointing for stochastic jobs with known execution time distributions. To the best of our knowledge, this is the first result to provide performance guarantee on the expected execution time while leveraging checkpointing in reservation-based scheduling environment.

Chapter 9

Reservation Strategies for Single Stochastic Job

Contents

9.1 Introduction

In this chapter, we introduce the notion of reservation strategies for stochastic jobs. After introducing a motivational example of the target problem, we present a formal definition of a stochastic job in Section 9.2. Then, Section 9.3 is dedicated to the presentation of a generic platform cost model. Finally, we state the optimization problem to be solved in Section 9.4.

9.1.1 Reservation-based approach

We are interested in processing a single stochastic job on a reservation-based platform. The notion of reservation-based platform is here very wide, and covers both cloud platforms (such as Amazon AWS[16]) as well as HPC infrastructures. We aim at providing solutions that cover a wide range of platforms and applications.

In general, scheduling a job onto such platforms typically involves making a reservation of the required resources, say of duration W_1 seconds, and running the job on the platform until either the job has successfully completed, or the reservation time has elapsed, whichever comes first.

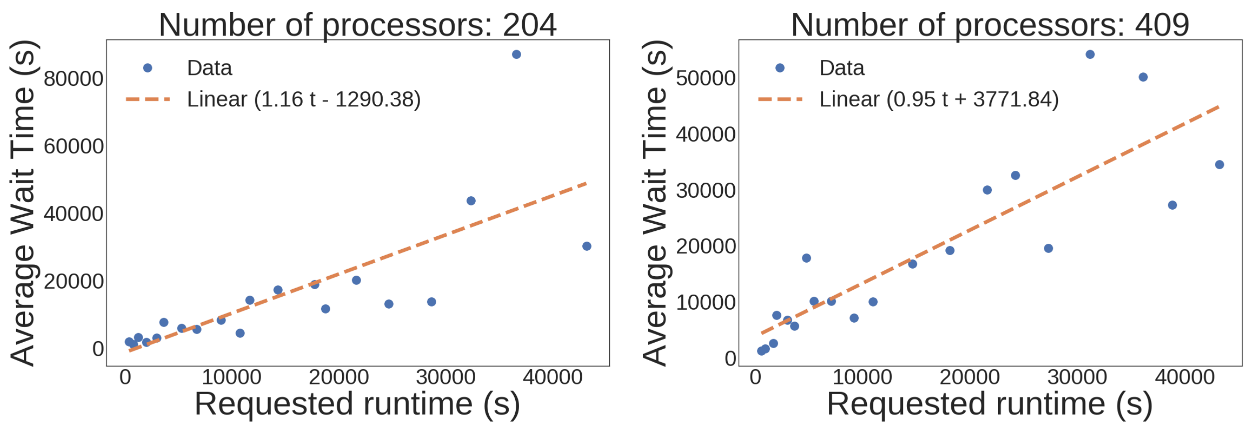
In the case of stochastic jobs, the user has to guess a good value for W_1. Indeed, if the job does not complete successfully within these W_1 seconds, the user has to resubmit the job, this time requiring a longer reservation, say of length $W_2 > W_1$. If the job still does not complete successfully within W_2 seconds, the user has to try again, using a reservation of length $W_3 > W_2$, and so on until the job would succeed eventually. The cost to the user is then the cost associated with all the reservations that were necessary to the successful completion of the job.

A reservation strategy could well depend on the context, the type of jobs and the platform. As an example, the MASI Lab [91] that we collaborate with at Vanderbilt University takes the average execution time from the last few instances of a neuroscience job to determine the first reservation time for its next instance. If the reservation is not enough, a standard practice is to resubmit the job by increasing with a factor 1.5 the requested time in the last failed run, some users tend to reserve a walltime that "guarantees" execution success (say up to the 99^{th} execution quantile). If this is not enough, they can ask for the 99^{th} execution quantile of the remaining possibilities, etc.

This reservation-based approach is agnostic of the type of the job (sequential or parallel; single task or workflow) and of the nature of the required computing resources (processors of a large supercomputer, virtual machines on a cloud platform, etc.). The user just needs to make good guesses for the values of successive reservation durations, hoping to minimize the associated cumulated cost. Here, we refer to cost as a generic metric. It could be paid either in terms of budget (e.g., a monetary amount as a function of what is requested and/or used in a cloud service), or in terms of time (e.g., the waiting time of the job in an HPC queue that depends on the requested runtime as shown in Figure 9.1).

Although we do not know the exact execution time of the job to be scheduled, we do not schedule completely in the dark. Instead, we assume that there are many jobs of similar type and that their execution times obey the same (known) probability distribution, that can be extracted from historic of past runs as we will see in next sections. Each job is deterministic, meaning that a second execution of the same job will last exactly as long as the first one. However, the exact execution time of a given job is not known until that job has successfully completed. Our only assumption is that job execution times are randomly and uniformly sampled from a target probability distribution.

We also include the possibility of checkpointing at the end of some (well-chosen) reservations. This allows us to save the current state of the application, and restart in a next reservation from that state. The idea of checkpointing is very natural and widely used in practice, in particular for long jobs lasting several hours, but it dramatically complicates the design of scheduling strategies. State-of-the-art approaches either



(a) Jobs that requested 204 procs.

(b) Jobs that requested 409 procs.

Figure 9.1: Average wait times of the jobs run on the same number of processors (204 and 409) as a function of the requested runtimes (data from [136]). All jobs are clustered into 20 groups, each with similar requested runtimes. Each point (in blue) shows the average wait time of all jobs in a group and the dotted lines (in orange) represent affine functions that fit the data.

checkpoint at the end of all reservations, or never checkpoint. For large-scale applications, checkpointing to save intermediate results at the end of each reservation is the de facto standard approach.

As checkpointing is not always supported for all applications (for instance, checkpointing applications running inside a container is currently under development [40]), we also provide solutions that do not model checkpointing. In this chapter, we introduce the model of reservations with possibility of checkpointing at the end of reservation. The model without checkpoint/restart is introduced in Section 10.2 of Chapter 10 before presenting the associated scheduling strategies.

While the core of the theoretical results presented in the remaining chapters of this manuscript is valid for general continuous probability distributions, we focus on the usual distributions for the evaluation. In particular, we consider Uniform, Beta, and Bounded Pareto distributions if the execution times are upper-bounded, i.e., they belong to some interval $[a, b]$; and we consider Exponential, Weibull, LogNormal, and a few others if there is no upper bound for the execution times. Note that the LogNormal distribution has been advocated to model file sizes [57], and we assume that job durations could naturally obey this distribution too. Also we only consider distributions whose support is included in $[0, \infty)$, because execution times must have positive values. This precludes the use of Normal distribution, for instance.

9.1.2 A generic cost function

The cost of a reservation is usually proportional to the reservation length, with a possible initial and fixed value (start-up overhead). In our will of targeting a wide range of platforms, we propose a generic cost model that fit both cloud/BigData and HPC infrastructures. A typical model of cloud platform is the *Reserved Instance* model available on Amazon AWS [16], which is up to 75% cheaper than the flexible *On-Demand* model that does not require advanced reservations. For HPC infrastructures, the cost of a reservation is similar, with a cost paid in proportion to the actual execution time, again with a possible start-up overhead. This latter scenario is relevant when submitting jobs to large supercomputing platforms, where each user requests a set of resources for a given number of hours, but only pays for the hours actually spent; however, the time spent in the waiting queue of the scheduler, and hence the job's waiting time, both depend upon the number of hours asked for in the request.

Specifically, for a reservation of length W_1 and an actual execution duration of length X, the cost is expressed as:

$$\alpha W_1 + \beta \min(W_1, X) + \gamma \tag{9.1}$$

where α, β and γ are constant parameters that depend on the platform and the cost model[1]. The first component αW_1 is proportional to the reservation length (pay for what you ask). The second component $\beta \min(W_1, X)$ is proportional to the actual execution time (pay for what you use). Finally, the third and last component is a start-up time possibly associated with the first and/or second components.

Altogether, this cost function covers many different cost models, going from execution time on an HPC infrastructure to cost when using virtual machines in Cloud Computing frameworks.

9.1.3 An illustrative example

We use an example to help understand the challenges of the problem under study. Consider the jobs depicted in Figure 9.2. We model their execution time with $\mathcal{D}$, a truncated LogNormal probability distribution on the domain $[a, b] = [0, 80\text{h}]$ (mean $\mu = 21$h, standard deviation $\sigma = 20$h). The exact execution time X of the next job to be scheduled is not known until that job has successfully completed, but instead is randomly and uniformly sampled from the target probability distribution $\mathcal{D}$. The objective is to minimize the expected cost of scheduling this job. To do so, we have to derive a sequence of reservations. Then we compute the cost of the job given that sequence, and aim at minimizing the expected value.

[1]Other cost functions could be envisioned. In particular, the cost for a reservation could be a more general function than a simple affine one. Most of the results in this manuscript can be extended to convex cost functions. When using a convex function impacts the theoretical results, we mention the associated modifications to be done. Basically, we focus on affine costs because of their wide applicability under various scenarios.

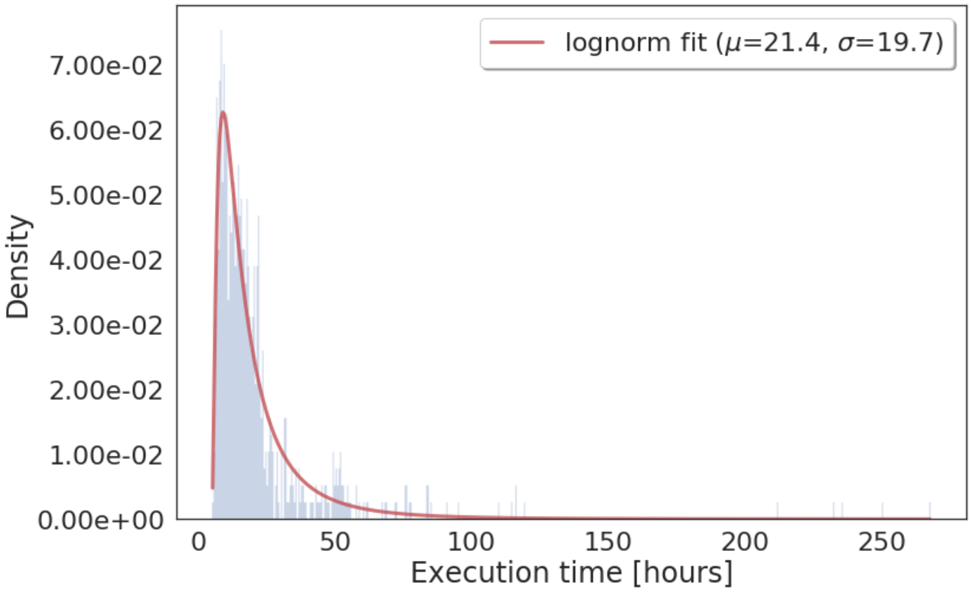


Figure 9.2: Execution times from 2017 for a *Structural identification of orbital anatomy* application, and its fitted distribution (in red).

We instantiate the cost model with $\alpha = 1, \beta = \gamma = 0$ in the example. In Figure 9.3, we depict three strategies, and their expected costs in hours: (i) S_1 (Standard), which reserves the upper bound of $\mathcal{D}$, $W_1 = b = 80$; (ii) S_2 (Without Checkpoint), which introduces a first reservation of size $W_1 = 20$ before the second reservation $W_2 = 80$; (iii) S_3 (With Checkpoint), which introduces a first checkpointed reservation of size $W_1 = 20 + 7$ (20 to cover jobs shorter than 20, and 7 (red box) is the cost to checkpoint), then a second non-checkpointed reservation of size $W_2 = 7 + 20$ (7 (green box) is the cost to restart, 20 to cover jobs larger than 20 and smaller than 40), and a third reservation of size $W_3 = 7 + 60$ (7 is the cost to restart, 60 to cover jobs of size up to b). Now, we show how to compute the expected cost of the different strategies. For S_1, there is a unique reservation that represents the total cost:

$$\mathbb{E}(S_1) = 80$$

For S_2, the expected cost is decomposed as follows. Either one reservation is sufficient to success, or the two reservations are needed. In the first case, the reservation W_1 of size 20 is performed and is weighted by $\mathbb{P}(X \leq 20)$, the probability of job success with this single reservation. In the second case, the first reservation has failed but has been paid, and one has to perform the second reservation $W_2 = 80$, for a total cost of $80 + 20 = 100$. This scenario occurs only if $20 < X \leq 80$. Hence, we have

$$\begin{aligned}\mathbb{E}(S_2) &= 20 \cdot \mathbb{P}(X \leq 20) + (80 + 20) \cdot \mathbb{P}(20 < X \leq 80) \\ &= 20 \times 0.66 + 100 \times 0.34 \\ &= 47.2\end{aligned}$$

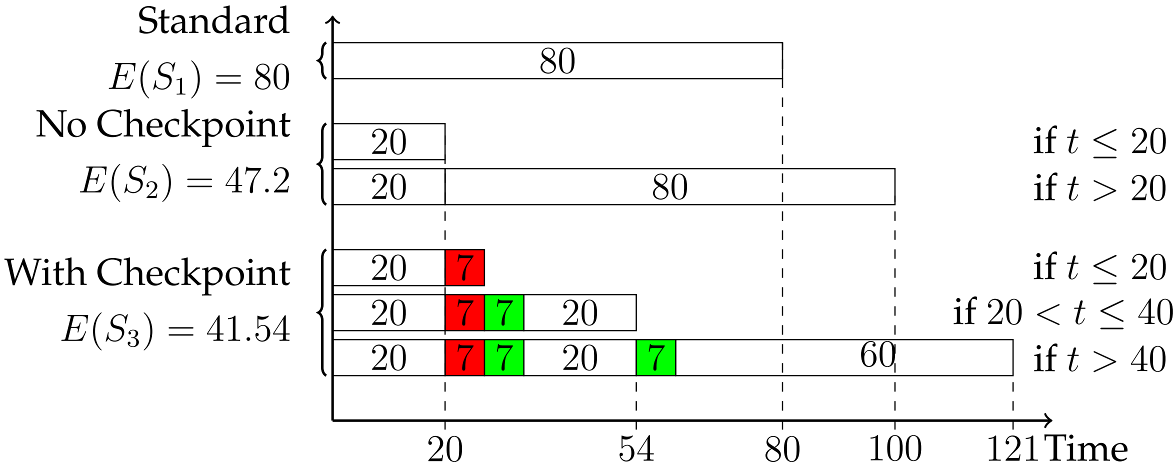


Figure 9.3: Illustration of different reservation strategies. The checkpoint (red) and restart (green) costs are equal to 7.

Regarding the expected cost of S_3, we apply the same principle as in the calculation of S_2. However, we leverage the wastage of failed reservations by using checkpointing (inducing an extra-cost in the reservation) and updating the probability associated to each reservation:

$$\begin{aligned}
\mathbb{E}(S_3) &= 27 \cdot \mathbb{P}\left(X \leq 20\right) + (27 + 27) \cdot \mathbb{P}\left(20 < X \leq 40\right) + (54 + 67) \cdot \mathbb{P}\left(40 < X\right) \\
&= 27 \times 0.66 + 54 \times 0.26 + 121 \times 0.08 \\
&= 41.54
\end{aligned}$$

Note that $\tilde{S}_3$, the variant of S_3 where the second reservation is also checkpointed, would have a larger expected cost due to this second checkpoint: $\mathbb{E}(\tilde{S}_3) = 27 \times 0.66 + 61 \times 0.26 + 128 \times 0.08 = 43.92$. Similarly one can verify that not performing the second reservation at all would also have increased the expected cost. This example shows that the duration of the reservations is a critical feature for performance. Also, we see that including checkpointing does help for some scenarios but has too much overhead for others, and suggests that finding the best trade-off is difficult.

Indeed, in the general case, one has to decide which reservations should be checkpointed, depending on application profile and platform parameters. Moreover, determining the expected cost of a given reservation sequence together with scheduling decisions gets quite complicated. In the remaining of this chapter, we introduce the formal definition of stochastic jobs, a formulation of the expected cost for a sequence of reservation as well as the optimization problem to solve.

9.2 Stochastic jobs

We consider stochastic jobs whose execution times are unknown but (i) deterministic with respect to input data, so that two successive executions of the same job will have the same duration; and (ii) randomly and uniformly sampled from a given probability distribution law $\mathcal{D}$, whose density function (PDF) is f and cumulative distribution function (CDF) is F. The probability distribution is assumed to be nonnegative, since we model execution times, and it is defined either on a finite support $[a, b]$, where $0 \leq a < b$, or on an infinite support $[a, \infty)$ where $a \geq 0$.

Hence, the execution time of a job is a random variable X, and

$$\mathbb{P}(X \leq T) = F(T) = \int_a^T f(t)dt$$

For notational convenience, we sometimes extend the domain of f outside the support of $\mathcal{D}$ by letting $f(t) = 0$ for $t \in [0, a] \cup [b, \infty)$.

These notations implies that $\mathcal{D}$ is a continuous distribution. At this point, it is necessary to explain the difference and benefits of using continuous or discrete probability distribution of the execution time. When an application is executed, it is straightforward to obtain historical data of the execution under the form of discrete values. From these historical data, two possibilities can be envisioned:

- use these raw data to generate an associated discrete probability distribution,
- interpolate a continuous distribution by fitting an usual and known distribution.

Intuitively, one could believe that discrete probability would be the most natural way to use historical data. However, recent work [59] shows that using continuous model by interpolating the historical data gives more robust solutions, especially when few data are available. Hence, we consider here a model to derive and evaluate candidate solutions from continuous distributions. However, we do not eliminate the possibility of using discrete distributions. In Chapter 10 that presents algorithms to derive reservation strategies from distributions, we also provide exact solutions in the case of discrete distribution. We also investigate solutions that discretize a continuous distribution into a discrete one, to use the exact associated algorithm. We compare the performance of all these heuristics in Chapter 11.

Finally, we assume that we can interrupt the jobs at any time (divisible load application) to take a checkpoint: this will save the current progress of the execution, and enable to restart from that point on. Divisible load applications can be found, for example, in biological computations, image and video processing [85]. Usually, a checkpoint is a snapshot of the memory state of the application at the time of the checkpoint. For now, we assume that the cost of checkpoint and of recovery are constant throughout the execution. Let C be the cost to checkpoint the data at the end of an execution, and R the cost to read the data to restart a computation.

Remark 1. *By setting either C or R to ∞, then it is never useful to checkpoint, hence this problem can be reduced to the problem without checkpointing presented in Section 10.2 of Chapter 10.*

9.3 Definitions and optimization problem

We present in this chapter the different notations that are necessary to define the problem under study.

9.3.1 General principles

We start by introducing the basic principles of reservation-based approach. To execute a job , the user makes a series of reservations, until the job successfully executes within the length of the last reservation. For a reservation of length W_1, and for an actual duration t of the job, the cost is $\alpha W_1 + \beta \min(W_1, t) + \gamma$, as stated in Equation (9.9), where $\alpha > 0$, $\beta \geq 0$ and $\gamma \geq 0$. If $W > W_1$, another reservation should be paid for.
Hence, the user needs to make a (possibly infinite) sequence of reservations $S = (W_1, W_2, \ldots, W_i, W_{i+1}, \ldots)$, where:

1. $W_i < W_{i+1}$ for all $i \geq 1$[2] Indeed, because jobs are deterministic, it is redundant to have a duration in the sequence that is not strictly larger than the previous one, hence that duration can be removed from the sequence;

2. all possible execution times of the job are indeed smaller than or equal to some W_i in the sequence. This simply means that the sequence must tend to infinity if job execution times are not upper-bounded.

We assume that both properties hold when speaking of a reservation sequence. For notational convenience, we define $W_0 = 0$, in order to simplify summations.

Moreover, we take the checkpointing process into account. If the job did not complete its execution during the last reservation, but was checkpointed during the last C seconds of that reservation, then in the current reservation, the job can restart from that checkpoint during the first R seconds, and then continue execution from its saved state. On the contrary, if no checkpoint was taken during the last reservation, the work done during that reservation is lost, and the execution must restart from the last checkpoint (or from the very beginning if no checkpoint was taken yet).

Altogether, the user needs to schedule a (possibly infinite) sequence of reservations and decide whether to take a checkpoint or not at the end of each reservation.

We use the cost model presented in Equation (9.1). For a reservation of length W and an actual execution duration w for the job, the cost is

$$\alpha W + \beta \min(W, w) + \gamma$$

where $\alpha > 0$, $\beta \geq 0$ and $\gamma \geq 0$.

[2]We considere here a sequence of reservations. This condition does not hold anymore when considering checkpointing, that we introduce in the next paragraphs.

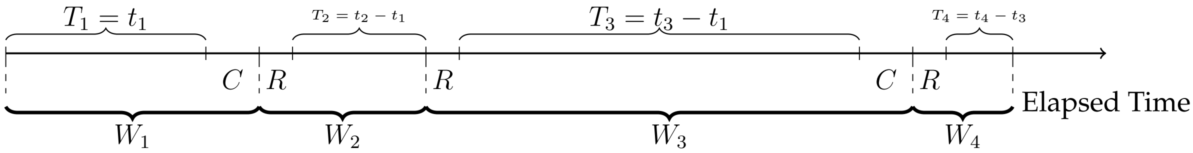


Figure 9.4: Graphical representation of elapsed time for the reservation sequence $\mathcal{S} = \{(W_1, 1), (W_2, 0), (W_3, 1), (W_4, 0)\}$.

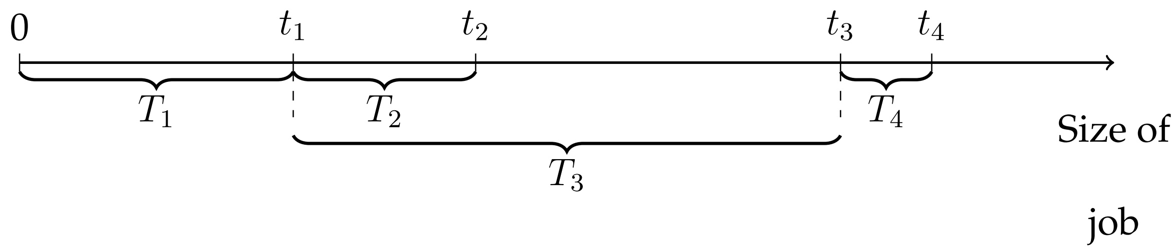


Figure 9.5: Graphical representation of job progress (and showing t_k versus T_k) for the reservation sequence $\mathcal{S} = \{(W_1, 1), (W_2, 0), (W_3, 1), (W_4, 0)\}$.

9.3.2 Notations and reservation-based strategies

We formally define what is a reservation sequence for an application whose execution time follows a probability distribution.

Definition 2 (Reservation sequence for $\mathcal{D}$)**.** Given a probability distribution $\mathcal{D}$, a *reservation sequence* $\mathcal{S} = \{(W_1, \delta_1), (W_2, \delta_2), \dots\}$, is defined as a sequence of reservation lengths W_k and a sequence of checkpointing decisions $\delta_k \in \{0, 1\}$: $\delta_k = 1$ means the k^{th} reservation ends with a checkpoint, and $\delta_k = 0$ means it does not.

Then, the k^{th} reservation can be decomposed into:

$$W_k = R_k + T_k + C_k \tag{9.2}$$

where R_k is the time spent for restart, T_k for actual job execution, and C_k for checkpoint. We have $C_k = \delta_k C$ by definition. There is a restart if and only if there has been a checkpoint at some point before, hence

$$R_k = (1 - \prod_{i=1}^{k-1}(1 - \delta_i))R$$

(assuming $R_1 = 0$ for the first reservation).

It is hard to keep track of actual job progress when using only the (W_k, δ_k) values. Consider for instance the following sequence $\mathcal{S} = \{(W_1, 1), (W_2, 0), (W_3, 1), (W_4, 0)\}$, which is depicted in Figure 9.4. If the actual job duration is $X = t$, during which reservation will the job complete its execution? We introduce another view of the reservation sequence $\mathcal{S}$ by introducing the milestones (t_k's) as shown in Figure 9.5. A milestone t_k represents the amount of work that has actually been executed at the end of the k^{th} reservation. Then, the last reservation for the job of length t is W_k, where $t_{k-1} \leq t \leq t_k$. Of course, we need that $t \leq t_4$ for all values of $\mathcal{D}$ (equivalently, the upper bound of the support of $\mathcal{D}$ is $b \leq t_4$) for all jobs to complete successfully with the four reservations of $\mathcal{S}$.

The relationship between the milestone t_k (actual work progress) and the value of T_k (time spent computing during reservation W_k; see Equation (9.2)) is the following:

$$t_k = T_k + \sum_{i=1}^{k-1} \delta_i T_i \tag{9.3}$$

Indeed, the work actually progresses only from the last checkpoint, while the work executed during the previous non-checkpointed reservations is lost whenever these non-checkpointed reservations do not allow for the full completion of the job. Another way to express the relationship between t_k and T_k is the following:

$$t_k = T_k + \max\{t_i \,\big|\, 1 \leq i \leq k-1 \text{ and } \delta_i = 1\} \tag{9.4}$$

Indeed, Equation (9.4) gives a recursive way to compute t_k from its definition. We recapitulate the relations between all notations introduced in Figures 9.4 and 9.5:

$$W_k = R_k + T_k + C_k \tag{9.5}$$
$$R_k = (1 - \Pi_{i<k}(1-\delta_i))R \tag{9.6}$$
$$\begin{aligned} T_k &= t_k - \sum_{i<k} \delta_i T_i \\ &= t_k - \max\{t_i \,\big|\, 1 \leq i \leq k-1 \text{ and } \delta_i = 1\} \end{aligned} \tag{9.7}$$
$$C_k = \delta_k C \tag{9.8}$$

In the following, we use milestones t_k rather than reservation lengths W_k to characterize a reservation sequence, and we write

$$\mathcal{S} = \{(t_1, \delta_1), (t_2, \delta_2), \dots\}$$

instead of

$$\mathcal{S} = \{(W_1, \delta_1), (W_2, \delta_2), \dots\}$$

because it is easier to use milestones when computing the expected cost of a sequence, as shown below. For notational convenience, we define $t_0 = 0$ as the first milestone of each sequence $\mathcal{S}$. Note also that we can restrict to sequences where $t_{k-1} < t_k$, because otherwise (if $t_{k-1} = t_k$), the execution does not progress during the k^{th} reservation.

From above notations, basic checkpointing policies can be described as follows:

- Checkpoint all reservations: $\forall \delta,\ \delta_i = 1$, leading to $t_i = T_i = W_i$
- Never Checkpoint: $\forall \delta,\ \delta_i = 0$, which implies that $t_i = \sum_{j \leq i} T_j$

9.4 Optimization problem

Given a reservation sequence $\mathcal{S} = \{(t_i, \delta_i)\}_i$ and a job with execution time t such that $t_{k-1} < t \leq t_k$, the cost of the sequence for that job is given by:

$$C_{\mathcal{S}}(k,t) = \sum_{i=1}^{k-1} (\alpha W_i + \beta W_i + \gamma) + \alpha W_k + \beta(R_k + t - (t_k - T_k)) + \gamma \tag{9.9}$$

where the first part is the total cost from the $k-1$ first reservations that did not allow the job to complete, and the second part is the cost of the k^{th} reservation. The actual execution time during the k^{th} reservation is $t - (t_k - T_k)$, because $t_k - T_k$ is the amount of work done up to the beginning of that reservation; we add the restart time (R_k) but do not need to checkpoint (if $\delta_k = 1$) because the job successfully completes before it is taken.

We let $k(t) = k$ for a job of length t such that $t_{k-1} < t \leq t_k$. Now, the expected cost of the reservation sequence $\mathcal{S}$ over a job whose execution time is a random variable X is

$$\mathbb{E}(\mathcal{S}(X)) = \int_0^\infty C_{\mathcal{S}}(k(t),t) f(t) dt = \sum_{k=1}^{\infty} \int_{t_{k-1}}^{t_k} C_{\mathcal{S}}(k,t) f(t) dt \tag{9.10}$$

We are now ready to state the optimization problem:

Definition 3 (STOCHASTIC-CKPT)**.** Given a random variable X following the distribution $\mathcal{D}$ (with PDF f and CDF F) for the execution times of stochastic jobs, and given a cost function given by Equation (9.9) (with parameters $\alpha > 0$, $\beta \geq 0$ and $\gamma \geq 0$), find a reservation strategy $\mathcal{S}$ with minimal expected cost $\mathbb{E}(\mathcal{S}(X))$ as given in Equation (9.10).

In the next chapter, we will use all these notations to derive solutions to STOCHASTIC-CKPT. We will also study a variant of the problem when checkpointing is not available.

Chapter 10

Algorithmic Solutions

Contents

In this chapter, we present algorithmic solutions to solve STOCHASTIC-CKPT problem, presented in Chapter 9.

We divide this algorithmic study in two sections. Section 10.1 presents the reservation strategies in the case of scheduling an application with the possibility of checkpointing at the end of some reservations (STOCHASTIC-CKPT, Problem 3 in Chapter 9). Section 10.2 presents model and solutions to the associated problem without checkpointing.

10.1 Solutions with checkpointing

In this section, we propose a solution to Problem 3 presented in Chapter 9. The main contributions for the scheduling study with checkpointing are the following:

- An arbitrarily close to optimal solution for any continuous probability distribution with bounded support, by providing a Fully Polynomial-Time Approximation Scheme (FPTAS) to compute a reservation sequence and its checkpointing decisions (Section 10.1.2).
- The characterization of an optimal reservation sequence, together with its checkpointing decisions, for any discrete probability distribution, using a sophisticated dynamic programming algorithm (Section 10.1.3).
- A set of heuristics to solve STOCHASTIC-CKPT for bounded continuous distributions (Section 10.1.4), based on extensions of the solutions presented in Section 10.1.2 and Section 10.1.3

Before presenting the different solutions, we first introduce some necessary notations.

10.1.1 Expected cost

We start by establishing a simpler expression for the expected cost function of STOCHASTIC-CKPT.

Theorem 4. *Given a random variable X and a reservation sequence $\mathcal{S} = \{(t_1, \delta_1), (t_2, \delta_2), \dots\}$, the expected cost $\mathbb{E}(\mathcal{S}(X))$ of a strategy $\mathcal{S}$ given by Equation (9.10), with parameters α, β and γ, can be rewritten as:*

$$\begin{aligned}\mathbb{E}(\mathcal{S}(X)) = \beta \cdot \mathbb{E}[X] + \alpha\big(t_1 + \delta_1 C\big) + \gamma \\ + \sum_{i=2}^{\infty} \Big(\alpha W_i + \beta\big(R_i + (1-\delta_{i-1})T_{i-1} + C_{i-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i-1}\right) \qquad (10.1)\end{aligned}$$

For simplicity, when there is no ambiguity on the random variable X, we denote $\mathbb{E}(\mathcal{S}(X)) = \mathbb{E}(\mathcal{S})$.

Proof. Firstly we rewrite Equation (10.1) as follows:

$$\mathbb{E}(\mathcal{S}) = \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{\infty} \Big(\alpha W_i + \beta\big(R_i + (1-\delta_{i-1})T_{i-1} + C_{i-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i-1}\right) \qquad (10.2)$$

with initialization $\delta_0 = W_0 = R_1 = 0$ and $t_0 = 0$.

From Equations (9.9) and (9.10), we have

$$\mathbb{E}(\mathcal{S}) = \mathbb{E}_1 + \mathbb{E}_2 + \mathbb{E}_3 \qquad (10.3)$$

where

$$\mathbb{E}_1 = \sum_{k=1}^{\infty} \int_{t_{k-1}}^{t_k} \Big(\sum_{i=1}^{k} (\alpha W_i + \gamma) \Big) f(t)dt$$
$$\mathbb{E}_2 = \sum_{k=1}^{\infty} \int_{t_{k-1}}^{t_k} \Big(\sum_{i=1}^{k-1} \beta W_i \Big) f(t)dt$$
$$\mathbb{E}_3 = \sum_{k=1}^{\infty} \int_{t_{k-1}}^{t_k} \beta \Big(t + R_k + T_k - t_k \Big) f(t)dt$$

Using $t_0 = 0$, we compute the first term as

$$\begin{aligned}
\mathbb{E}_1 &= \sum_{k=1}^{\infty} \sum_{i=1}^{k} (\alpha W_i + \gamma) \int_{t_{k-1}}^{t_k} f(t)dt \\
&= \sum_{k=1}^{\infty} \sum_{i=1}^{k} (\alpha W_i + \gamma) \cdot \mathbb{P}\left(t_{k-1} < X \leq t_k\right) \\
&= \sum_{i=1}^{\infty} (\alpha W_i + \gamma) \sum_{k=i}^{\infty} \mathbb{P}\left(t_{k-1} < X \leq t_k\right) \\
&= \sum_{i=1}^{\infty} (\alpha W_i + \gamma) \cdot \mathbb{P}\left(X > t_{i-1}\right)
\end{aligned}$$

Similarly, using $W_0 = 0$, we express the second term as:

$$\mathbb{E}_2 = \sum_{i=1}^{\infty} \beta W_i \cdot \mathbb{P}\left(X > t_i\right)$$

Finally, we derive the third term as:

$$\begin{aligned}
\mathbb{E}_3 &= \int_{t_0}^{\infty} \beta t f(t)dt + \sum_{k=1}^{\infty} \int_{t_{k-1}}^{t_k} \beta \Big(R_k + T_k - t_k \Big) f(t)dt \\
&= \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{\infty} \beta \Big(R_i + T_i - t_i \Big) \cdot \mathbb{P}\left(t_{i-1} < X \leq t_i\right)
\end{aligned}$$

Plugging these three terms back into Equation (10.3), we get:

$$\begin{aligned}\mathbb{E}(\mathcal{S}) = \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{\infty} (\alpha W_i + \beta W_{i-1} + \gamma) \cdot \mathbb{P}\left(X > t_{i-1}\right) \\ + \sum_{i=1}^{\infty} \beta \left(R_i + T_i - t_i\right) \cdot \mathbb{P}\left(t_{i-1} < X \leq t_i\right) \\ = \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{\infty} (\alpha W_i + \beta(R_i + T_{i-1} + C_{i-1}) + \gamma) \cdot \mathbb{P}\left(X > t_{i-1}\right) \\ - \sum_{i=1}^{\infty} \beta \left(t_i - T_i\right) \cdot \mathbb{P}\left(t_{i-1} < X \leq t_i\right)\end{aligned} \tag{10.4}$$

For the last derivation, we used

$$\sum_{i=1}^{\infty} (R_{i-1} \cdot \mathbb{P}\left(X > t_{i-1}\right) + R_i \cdot \mathbb{P}\left(t_{i-1} < X \leq t_i\right)) = \sum_{i=1}^{\infty} (R_i \cdot \mathbb{P}\left(X > t_{i-1}\right)$$

Now, we study the second part of Equation (10.4) above. For all $j \leq 1$, let $\phi^o(j)$ denote the index of the j^{th} checkpointed reservation of $\mathcal{S}$. For instance in the example of Figures 9.4 and 9.5, $\phi^o(1) = 1$ and $\phi^o(2) = 3$. Then, using Equation (9.7), we have

$$\begin{aligned}\sum_{i=1}^{\infty} (t_i - T_i) \cdot \mathbb{P}\left(t_{i-1} < X \leq t_i\right) &= \sum_{j=1}^{\infty} t_{\phi(j)} \cdot \mathbb{P}\left(t_{\phi(j)} < X \leq t_{\phi(j+1)}\right) \\ &= \sum_{j=1}^{\infty} T_{\phi(j)} \cdot \mathbb{P}\left(X > t_{\phi(j)}\right) \\ &= \sum_{i=1}^{\infty} \delta_i T_i \cdot \mathbb{P}\left(X > t_i\right)\end{aligned}$$

Plugging the above back into Equation (10.4), we get the desired result shown in Equation (10.2). □

10.1.2 Execution time as a continuous probability distribution

In this section, we provide an approximation algorithm of the optimal strategy for continuous distributions with bounded support $[a, b]$, where $a \geq 0$ and b is finite. Because we model job execution times, it is natural to truncate continuous distributions whose support is $[0, \infty[$ such as an Exponential or LogNormal distribution, say, to a bounded support $[a, b]$.

The result for continuous distribution is particularly important: recent work [59] have shown that continuous distributions gave strategies that allowed using small data samples to find an efficient strategy. Here, it returns an arbitrarily good quality solution with low complexity.

More precisely, let X be a continuous random variable defined on $[a,b]$ modeling the probability distribution $\mathcal{D}$, where $0 \leq a < b$, with CDF F and PDF f. We propose a solution for STOCHASTIC-CKPT under the form of a fully polynomial-time approximation scheme. Before presenting the solution and its proof, we start by showing the following Lemma:

Lemma 1. *Given a random variable X and a strategy $\mathcal{S} = \{(t_1, \delta_1), \dots, (t_{|S|}, \delta_{|S|})\}$, if there exists an index $i_0 > 1$ such that $t_1 < \cdots < t_{i_0-1} \leq \min(R, \varepsilon\mathbb{E}[X]) < t_{i_0} < \cdots < t_{|S|}$, then the strategy $\tilde{\mathcal{S}} = \{(\min(R, \varepsilon\mathbb{E}[X]), 0), (t_{i_0}, \delta_{i_0}), \dots, (t_{|S|}, \delta_{|S|})\}$ satisfies:*

$$\mathbb{E}(\tilde{\mathcal{S}}(X)) \leq (1+\varepsilon) \cdot \mathbb{E}(\mathcal{S}(X))$$

Intuitively, this lemma states that restricting to strategies such that the first reservation is at least $\min(R, \varepsilon\mathbb{E}[X])$ only increases the cost by at most a factor of $1+\varepsilon$.

Proof. Consider a strategy $\mathcal{S} = \{(t_1, \delta_1), \dots, (t_{|S|}, \delta_{|S|})\}$ for a random variable X, such that there exists an index $i_0 > 1$ with

$$t_1 < \cdots < t_{i_0-1} \leq \min(R, \varepsilon\mathbb{E}[X]) < t_{i_0} < \cdots < t_{|S|}$$

For simplicity, we denote by $\tilde{a} = \min(R, \varepsilon\mathbb{E}[X])$, and define strategy $\tilde{\mathcal{S}} = \{(\tilde{a}, 0), (t_{i_0}, \delta_{i_0}), \dots, (t_{|S|}, \delta_{|S|})\}$.

From Equation (10.1) we have:

$$\begin{aligned}
\mathbb{E}(\mathcal{S}(X)) \geq\ & \beta \cdot \mathbb{E}[X] \\
& + \alpha(t_1 + C_1) + \gamma \\
& + \Big(\alpha W_{i_0} + \beta\big(R_{i_0} + (1-\delta_{i_0-1})T_{i_0-1} + C_{i_0-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i_0-1}\right) \\
& + \sum_{i=i_0+1}^{|\mathcal{S}|} \Big(\alpha W_i + \beta\big(R_i + (1-\delta_{i-1})T_{i-1} + C_{i-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i-1}\right) \\
\mathbb{E}(\tilde{\mathcal{S}}(X)) =\ & \beta \cdot \mathbb{E}[X] \\
& + (\alpha\tilde{a} + \gamma) + (\alpha\tilde{W}_{i_0} + \beta\tilde{a} + \gamma) \cdot \mathbb{P}\left(X > \tilde{a}\right) \\
& + \sum_{i=i_0+1}^{|\mathcal{S}|} \Big(\alpha \tilde{W}_i + \beta\big(\tilde{R}_i + (1-\delta_{i-1})\tilde{T}_{i-1} + \tilde{C}_{i-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i-1}\right)
\end{aligned}$$

We obviously have $C_i = \tilde{C}_i,\ \forall i \geq i_0$.

We now show the following property:

$$\forall i \geq i_0,\ W_i \geq \tilde{W}_i$$

To show this, we consider two cases:

1. the last checkpoint before t_i was done during t_j with $j \geq i_0$ or there was no checkpoint before t_i.
   In this case, we obviously have $W_i = \tilde{W}_i$;

2. the last checkpoint before t_i was done during t_j with $j < i_0$ in $\mathcal{S}$, and there was no checkpoint done in $\tilde{\mathcal{S}}$ before t_i.

In this case, we have $W_i = R + (t_i - t_j) + \delta_i C$ and $\tilde{W}_i = t_i + \delta_i C$.
Since $t_j \leq t_{i_0 - 1} \leq R$, we get $W_i \geq \tilde{W}_i$.
Similarly, we can show that, $\forall i \geq i_0$, $R_i \geq \tilde{R}_i$.

Further, since $\mathbb{P}\left(X > \tilde{a}\right) \leq \mathbb{P}\left(X > t_{i_0-1}\right) \leq 1$, we can now derive that:

$$
\begin{aligned}
\mathbb{E}(\tilde{\mathcal{S}}(X)) - \mathbb{E}(\mathcal{S}(X)) &\leq \alpha(\tilde{a} - t_1 - C_1) + \beta(\tilde{a} - R_{i_0} - (1 - \delta_{i_0-1})T_{i_0-1} - C_{i_0-1}) \cdot \mathbb{P}\left(X > t_{i_0-1}\right) \\
&\leq (\alpha + \beta)\tilde{a} \\
&\leq \varepsilon(\alpha + \beta)\mathbb{E}[X]
\end{aligned}
$$

Finally, note that we immediately have $\mathbb{E}(\mathcal{S}(X)) \geq (\alpha + \beta)\mathbb{E}[X] + \gamma$, because this is the cost of an omniscient strategy that makes a single reservation of exactly the right size for each job. Therefore, we get the result:

$$
\mathbb{E}(\tilde{\mathcal{S}}(X)) - \mathbb{E}(\mathcal{S}(X)) \leq \varepsilon \cdot \mathbb{E}(\mathcal{S}(X))
$$

which completes the proof of Lemma 1. □

Armed of this lemma, we now present Algorithm 4 that computes a solution to STOCHASTIC-CKPT with a complexity $\mathcal{O}(\frac{1}{\varepsilon^3})$. Then, we introduce Theorem 5 showing that Algorithm 4 computes a fully polynomial-time approximation scheme for STOCHASTIC-CKPT.

Algorithm 4 DYN-PROG-COUNT(X, ε)

1: Let $[a, b]$ be the domain of X, with $0 \leq a < b$
2: $c_0 = 3(b - a)\min\left(\frac{1}{\min(\max(a, \varepsilon\mathbb{E}[X]/3), R, C)}, \frac{\alpha+\beta}{\gamma}\right)$
3: $n \leftarrow \lceil c_0/\varepsilon \rceil$
4: Define the discrete distribution $Y_n \sim (v_i, f_i)_{i=1\ldots n}$ s.t.

$$
\begin{cases}
v_i & = a + i \cdot \frac{b-a}{n} \qquad\qquad\qquad\qquad \text{for } 0 \leq i \leq n \\
f_i & = \mathbb{P}\left(Y_n = v_i\right) = \mathbb{P}\left(v_{i-1} < X \leq v_i\right) \quad \text{for } 1 \leq i \leq n
\end{cases}
$$

5: $\mathcal{S}_n^{\text{dp}} \leftarrow$ Optimal strategy for Y_n (Theorem 6 in Section 10.1.3 for discrete distribution)
6: **return** $\mathcal{S}_n^{\text{dp}}$

Theorem 5. *Given a continuous random variable X on the domain $[a,b]$, where $0 \le a < b$, and given a constant $\varepsilon > 0$,* DYN-PROG-COUNT(X,ε) *is a $1+\varepsilon$-approximation algorithm for* STOCHASTIC-CKPT *and executes in time $\mathcal{O}\big(\frac{1}{\varepsilon^3}\big)$.*

Proof. Given a continuous random variable X of support $[a,b]$, we define the discrete random variable $Y_n \sim (v_i, f_i)_{i=1\dots n}$ as stated in Algorithm 4:

$$\begin{cases} v_i & = a + i \cdot \frac{b-a}{n} & \text{for } 0 \le i \le n \\ f_i & = \mathbb{P}\left(Y_n = v_i\right) = \mathbb{P}\left(v_{i-1} < X \le v_i\right) & \text{for } 1 \le i \le n \end{cases} \tag{10.5}$$

Let $\mathcal{S}^{\text{opt}} = \{(\tilde{t}_i^o, \tilde{\delta}_i^o)\}_{1 \le i \le |\mathcal{S}^{\text{opt}}|}$ denote the optimal solution for X, and let $\mathcal{S}_n^{\text{dp}}$ denote the optimal solution for Y_n returned by Theorem 6, presented in Section 10.1.3, that corresponds to an optimal solution any discrete distributions. We want to show that

$$\mathbb{E}(\mathcal{S}_n^{\text{dp}}(X)) \le (1+\varepsilon) \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}(X))$$

In order to do that, we construct two intermediate strategies $\mathcal{S}_{\varepsilon/3}^{\text{opt}}$ and $\mathcal{S}^{\text{algo}}$ as follows.

First, $\mathcal{S}_{\varepsilon/3}^{\text{opt}} = \left((t_i^o, \delta_i^o)\right)_i$ is constructed in such a way that if $\tilde{t}_1^o \ge \min(R, \frac{\varepsilon \mathbb{E}[X]}{3})$, then $\mathcal{S}_{\varepsilon/3}^{\text{opt}} = \mathcal{S}^{\text{opt}}$, otherwise we construct $\mathcal{S}_{\varepsilon/3}^{\text{opt}}$ from $\mathcal{S}^{\text{opt}}$ by following Lemma 1 (with the value $\frac{\varepsilon}{3}$). Then, according to Lemma 1, we have:

$$\mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X)) \le \left(1 + \frac{\varepsilon}{3}\right) \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}(X)) \tag{10.6}$$

Second, $\mathcal{S}^{\text{algo}} = ((t_i^a, \delta_i^a))_{1 \le i \le |\mathcal{S}_{\varepsilon/3}^{\text{opt}}|}$ (hence $|\mathcal{S}^{\text{algo}}| = |\mathcal{S}_{\varepsilon/3}^{\text{opt}}|$), is such that for $1 \le i \le |\mathcal{S}_{\varepsilon/3}^{\text{opt}}|$, we let $(t_i^a, \delta_i^a) = (v_{\pi^o(i)}, \delta_i^o)$. Here, we use the sequence $(v_i)_{i=0\dots n}$ from Equation (10.5), and the function π^o defined below:

$$v_{\pi^o(i)-1} < t_i^o \le v_{\pi^o(i)} \tag{10.7}$$

In other words, for each reservation, $\mathcal{S}^{\text{algo}}$ chooses the first discrete value larger than or equal to the corresponding one chosen by $\mathcal{S}_{\varepsilon/3}^{\text{opt}}$, and makes the same checkpointing decision.

Lemma 2. $\mathbb{E}(\mathcal{S}^{\text{algo}}(X)) \le (1 + \frac{\varepsilon}{3}) \cdot \mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X))$.

Proof. We use the notations T_i^o, R_i^o, C_i^o, W_i^o for the parameters of $\mathcal{S}_{\varepsilon/3}^{\text{opt}}$, and T_i^a, R_i^a, C_i^a, W_i^a for the parameters of $\mathcal{S}^{\text{algo}}$. From Equations (9.5) to (9.8), we see that, for $1 \le i \le |\mathcal{S}_{\varepsilon/3}^{\text{opt}}|$, we have:

1. $\delta_i^o = \delta_i^a$;

2. $R_i^o = R_i^a$;

3. $C_i^o = C_i^a$; and
4. $W_i^a - W_i^o = T_i^a - T_i^o$.

In addition, if $\sigma^o(i)$ (resp. $\sigma^a(i)$) is the index of the last checkpoint before t_i^o (resp. t_i^a), then $\sigma^o(i) = \sigma^a(i)$, and,

$$\begin{aligned}
|T_i^a - T_i^o| &= \left|\left(t_i^a - t_{\sigma^a(i)}^a\right) - \left(t_i^o - t_{\sigma^o(i)}^o\right)\right| \\
&= \left|\left(v_{\pi^o(i)} - v_{\pi^o(\sigma^o(i))}\right) - \left(t_i^o - t_{\sigma^o(i)}^o\right)\right| \\
&= \left|\left(v_{\pi^o(i)} - t_i^o\right) - \left(v_{\pi(\sigma^o(i))} - t_{\sigma^o(i)}^o\right)\right| \\
&\leq \max\left(v_{\pi^o(i)} - t_i^o, v_{\pi^o(\sigma^o(i))} - t_{\sigma^o(i)}^o\right) \leq \frac{b-a}{n}
\end{aligned}$$

From Equation (10.1) we have:

$$\mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X)) = \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|} \left(\alpha W_i^o + \beta\big(R_i^o + (1-\delta_{i-1}^o)T_{i-1}^o + C_{i-1}^o\big) + \gamma\right) \cdot \mathbb{P}\left(X > t_{i-1}^o\right)$$

$$\mathbb{E}(\mathcal{S}^{\text{algo}}(X)) = \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|} \left(\alpha W_i^a + \beta\big(R_i^a + (1-\delta_{i-1}^a)T_{i-1}^a + C_{i-1}^a\big) + \gamma\right) \cdot \mathbb{P}\left(X > t_{i-1}^a\right)$$

We first observe that $\mathbb{P}\left(X > t_{i-1}^a\right) \leq \mathbb{P}\left(X > t_{i-1}^o\right)$ because $t_{i-1}^a \geq t_{i-1}^o$. We can derive that:

$$\begin{aligned}
\mathbb{E}(\mathcal{S}^{\text{algo}}(X)) - \mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X)) &\leq \sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|} \left(\alpha|T_i^a - T_i^o| + \beta(1-\delta_{i-1}^o)|T_{i-1}^a - T_{i-1}^o|\right) \cdot \mathbb{P}\left(X > t_{i-1}^o\right) \\
&\leq \alpha\frac{b-a}{n} + \sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|-1} \left(\left(\alpha + \beta(1-\delta_i^o)\right)\frac{b-a}{n}\right) \cdot \mathbb{P}\left(X > t_i^o\right) \\
&\leq \frac{b-a}{n}\left(\alpha + (\alpha+\beta)\sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|-1} \mathbb{P}\left(X > t_i^o\right)\right)
\end{aligned}$$

We also observe that:

$$\mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X)) \geq \gamma + \sum_{i=1}^{|\mathcal{S}_{\varepsilon/3}^{\text{opt}}|-1} \gamma \cdot \mathbb{P}\left(X > t_i^o\right)$$

Further, for $1 \leq i \leq |\mathcal{S}_{\varepsilon/3}^{\text{opt}}|$, we have $W_i^o \geq R_i^o + T_i^o \geq \min(R, \tilde{a})$, where $\tilde{a} = \max(a, \min(R, \varepsilon\mathbb{E}[X]/3))$. This is because either $T_i^o \geq \tilde{a}$ according to Lemma 1 (when there was no checkpoint before t_i^o), or $R_i^o = R$ (when there was a checkpoint before t_i^o). Therefore, we can derive:

$$\mathbb{E}(\mathcal{S}^{\text{opt}}_{\varepsilon/3}(X)) \geq \min(\max(a, \varepsilon\mathbb{E}[X]/3), R, C)\left(\alpha + (\alpha+\beta)\sum_{i=1}^{|\mathcal{S}^{\text{opt}}_{\varepsilon/3}|-1} \mathbb{P}\left(X > t_i^o\right)\right)$$

Note that

$$\min(R, \max(a, \min(R, \varepsilon\mathbb{E}[X]/3))) = \min(\max(a, \varepsilon\mathbb{E}[X]/3), R)$$

Using the definition of $c_0 = 3(b-a)\min\left(\frac{1}{\min(\max(a,\varepsilon\mathbb{E}[X]/3),R,C)}, \frac{\alpha+\beta}{\gamma}\right)$ in Algorithm 4, we obtain:

$$\mathbb{E}(\mathcal{S}^{\text{algo}}(X)) - \mathbb{E}(\mathcal{S}^{\text{opt}}_{\varepsilon/3}(X)) \leq \frac{c_0}{n} \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}_{\varepsilon/3}(X)) \leq \frac{\varepsilon}{3} \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}_{\varepsilon/3}(X))$$

which concludes the proof of Lemma 2. $\square$

Lemma 3. $\mathbb{E}(\mathcal{S}^{\text{dp}}_n(X)) \leq \mathbb{E}(\mathcal{S}^{\text{algo}}(X))$

Proof. Given any reservation strategy $\mathcal{S} = \{(t_i, \delta_i)\}_{1\leq i\leq|\mathcal{S}|}$ such that $\forall i,\ t_i \in \{v_1, \ldots, v_n\}$, we show that:

$$\mathbb{E}(\mathcal{S}(Y_n)) - \mathbb{E}(\mathcal{S}(X)) = \beta\left(\mathbb{E}[Y_n] - \mathbb{E}[X]\right)$$

Indeed, for the two distributions Y_n and X, the only differences in the cost function are: (i) the expectations $\mathbb{E}[Y_n]$ and $\mathbb{E}[X]$; and (ii) the probability values $\mathbb{P}\left(Y_n > t_i\right)$ and $\mathbb{P}\left(X > t_i\right), \forall i$.
But if $t_i \in \{v_1, \ldots, v_n\}$, we have:

$$\begin{aligned}
\mathbb{P}\left(Y_n > t_i\right) &= \mathbb{P}\left(Y_n > v_k\right) \\
&= \mathbb{P}\left(Y_n \in \cup_{j=k+1}^{n}\{v_j\}\right) = \sum_{j=k+1}^{n} \mathbb{P}\left(Y_n = v_j\right) \\
&= \sum_{j=k+1}^{n} \mathbb{P}\left(X \in]v_{j-1}, v_j]\right) = \mathbb{P}\left(X \in]v_k, v_n]\right) \\
&= \mathbb{P}\left(X > v_k\right) = \mathbb{P}\left(X > t_i\right)
\end{aligned}$$

We obtain that:

$$\mathbb{E}(\mathcal{S}(Y_n)) - \mathbb{E}(\mathcal{S}(X)) = \beta\left(\mathbb{E}[Y_n] - \mathbb{E}[X]\right)$$

We apply this result to both $\mathcal{S}^{\text{dp}}_n$ and $\mathcal{S}^{\text{algo}}$ and derive that:

$$\mathbb{E}(\mathcal{S}^{\text{dp}}_n(Y_n)) - \mathbb{E}(\mathcal{S}^{\text{dp}}_n(X)) = \mathbb{E}(\mathcal{S}^{\text{algo}}(Y_n)) - \mathbb{E}(\mathcal{S}^{\text{algo}}(X))$$

or equivalently,

$$\mathbb{E}(\mathcal{S}^{\text{dp}}_n(Y_n)) - \mathbb{E}(\mathcal{S}^{\text{algo}}(Y_n)) = \mathbb{E}(\mathcal{S}^{\text{dp}}_n(X)) - \mathbb{E}(\mathcal{S}^{\text{algo}}(X))$$

But $\mathcal{S}_n^{\text{dp}}$ is optimal for Y_n, hence

$$\mathbb{E}(\mathcal{S}_n^{\text{dp}}(Y_n)) - \mathbb{E}(\mathcal{S}^{\text{algo}}(Y_n)) \leq 0$$

Therefore,

$$\mathbb{E}(\mathcal{S}_n^{\text{dp}}(X)) - \mathbb{E}(\mathcal{S}^{\text{algo}}(X)) \leq 0$$

This concludes the proof of Lemma 3. □

Now, combining Lemma 2, Lemma 3 and Equation (10.6), we get:

$$\begin{aligned}
\mathbb{E}(\mathcal{S}_n^{\text{dp}}(X)) &\leq \mathbb{E}(\mathcal{S}^{\text{algo}}(X)) \\
&\leq \left(1+\frac{\varepsilon}{3}\right) \cdot \mathbb{E}(\mathcal{S}_{\varepsilon/3}^{\text{opt}}(X)) \\
&\leq \left(1+\frac{\varepsilon}{3}\right)\left(1+\frac{\varepsilon}{3}\right) \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}(X)) \\
&\leq (1+\varepsilon) \cdot \mathbb{E}(\mathcal{S}^{\text{opt}}(X))
\end{aligned}$$

which concludes the proof of Theorem 5. □

To conclude, we presented a near-optimal solution for STOCHASTIC-CKPT problem. This result theoretically gives strong guarantee on performance and is an important contribution to the problem under study.

10.1.3 Execution time as a discrete probability distribution

In this section, we provide an optimal algorithm for walltime given as a discrete probability distribution.

An optimal dynamic-programming algorithm

We now study Problem 3 for a finite discrete distribution:

$$Y \sim (v_i, f_i)_{1 \leq i \leq n}$$

where $v_i < v_{i+1}$ for all $1 \leq i \leq n-1$ and $f_i = \mathbb{P}(Y = v_i)$. We assume that $f_n \neq 0$ and $\sum_{i=1}^{n} f_i = 1$.

Consider a strategy $\mathcal{S} = \{(t_1, \delta_1), (t_2, \delta_2), \ldots, (t_{|\mathcal{S}|}, \delta_{|\mathcal{S}|})\}$, where $t_i = v_{\pi(i)}$ and $t_i < t_{i+1}$ for all $1 \leq i \leq |\mathcal{S}| - 1$. Also, the last reservation is necessarily $t_{|\mathcal{S}|} = v_n$ to ensure that the expected cost of the strategy is finite. By convention, we let $t_0 = v_0 = a$, hence $\mathbb{P}(Y > t_0) = 1$. Note that we can safely restrict to strategies where each milestone t_i is equal to some threshold v_j of the discrete distribution: otherwise, replacing t_i by the largest v_j such that $v_j \leq t_i$ leads to a smaller cost.

Rewriting Equation (10.1) with $W_i = R_i + T_i + C_i$, and since $W_0 = 0$, the expected cost of strategy $\mathcal{S}$ can be expressed as:

$$\begin{aligned}\mathbb{E}(\mathcal{S}) = {} & \beta \cdot \mathbb{E}[Y] \\ & + \sum_{i=1}^{|\mathcal{S}|} \big(\alpha\left(R_i + T_i + C_i\right) + \beta R_i + \gamma\big) \cdot \mathbb{P}\left(Y > t_{i-1}\right) \\ & + \sum_{i=1}^{|\mathcal{S}|-1} \beta\big((1-\delta_i)T_i + C_i\big) \cdot \mathbb{P}\left(Y > t_i\right)\end{aligned} \tag{10.8}$$

Based on Equation (10.8), and using Equations (9.6) to (9.8), we construct a dynamic programming algorithm to compute the optimal reservation sequence, that we name DISCRETE-CKPT:

Theorem 6. *For a discrete distribution $Y \sim (v_i, f_i)_{1\le i\le n}$, the optimal expected cost is returned by $\mathbb{E}_{\text{ckpt}}(0,0)$, where, for $0 \le i_c \le i_l \le n$, $\mathbb{E}_{\text{ckpt}}(i_c, i_l)$ is:*

$$\begin{aligned} &= \beta \cdot \mathbb{E}[Y], \qquad \textit{if } i_l = n \\ &= \min_{\substack{i_l+1\le j\le n,\\ \Delta_j\in\{0,1\}}} \left(\mathbb{E}_{\text{ckpt}}(\Delta_j j, j) + \Big(\alpha\big(v_j + \Delta_j C\big) + \gamma\Big) \cdot \sum_{k=i_l+1}^{n} f_k + \beta\big((1-\Delta_j)v_j + \Delta_j C\big) \cdot \sum_{k=j+1}^{n} f_k \right) \\ & \qquad \textit{if } i_c = 0 \\ &= \min_{\substack{i_l+1\le j\le n,\\ \Delta_j\in\{0,1\}}} \left(\mathbb{E}_{\text{ckpt}}\big((1-\Delta_j)i_c + \Delta_j j, j\big) + \Big(\alpha\big(R + (v_j - v_{i_c}) + \Delta_j C\big) + \beta R + \gamma\Big) \cdot \sum_{k=i_l+1}^{n} f_k \right. \\ & \quad \left. + \beta\big((1-\Delta_j)(v_j - v_{i_c}) + \Delta_j C\big) \cdot \sum_{k=j+1}^{n} f_k \right), \qquad \textit{otherwise} \end{aligned}$$

The optimal solution can be computed in $O(n^3)$ time.

Intuitively, i_c denotes the index of the last checkpointed value, while i_l denotes the index of the last value that was tried before we try the next one with index j. Here, Δ_j indicates whether the value v_j will be checkpointed or not.

Proof. To prove the optimality, consider $\mathbb{E}(\mathcal{S})$ given in Equation (10.8) for any reservation sequence:

$$\begin{aligned}\mathcal{S} &= \{(t_1, \delta_1), \ldots, (t_{|\mathcal{S}|}, \delta_{|\mathcal{S}|})\} \\ &= \{(v_{\pi(1)}, \Delta_{\pi(1)}), \ldots, (v_{\pi(|\mathcal{S}|)}, \Delta_{\pi(|\mathcal{S}|)})\}\end{aligned}$$

and define $\mathbb{E}_\ell$ as the following partial sum:

$$\begin{aligned}\mathbb{E}_\ell &= \beta \cdot \mathbb{E}[Y] \\ &+ \sum_{i=\ell+1}^{|\mathcal{S}|} \big(\alpha\left(R_i + T_i + C_i\right) + \beta R_i + \gamma\big) \cdot \mathbb{P}\left(Y > t_{i-1}\right) \\ &+ \sum_{i=\ell+1}^{|\mathcal{S}|-1} \beta\big((1-\delta_i)T_i + C_i\big) \cdot \mathbb{P}\left(Y > t_i\right)\end{aligned} \tag{10.9}$$

Note that $\mathbb{E}_0 = \mathbb{E}(\mathcal{S})$. We show by induction that the following invariant is true for all $\ell = |\mathcal{S}|, |\mathcal{S}|-1, \ldots, 0$:

$$\mathbb{E}_\ell \geq \mathbb{E}_{\text{ckpt}}(\pi(\bar{\ell}), \pi(\ell)) \tag{10.10}$$

where $\pi(\bar{\ell})$ is the index of the last reservation not larger than $\pi(\ell)$ and such that $\Delta_{\pi(\bar{\ell})} = 1$. We denote the corresponding reservation by $t_{\bar{\ell}} = v_{\pi(\bar{\ell})}$.

For the base case with $\ell = |\mathcal{S}|$, we have $\pi(|\mathcal{S}|) = n$, and $\mathbb{E}_{|\mathcal{S}|} = \beta \cdot \mathbb{E}[Y] = \mathbb{E}_{\text{ckpt}}(\pi(\overline{|\mathcal{S}|}), n)$. Now, suppose $\mathbb{E}_{\ell+1} \geq \mathbb{E}_{\text{ckpt}}(\pi(\overline{\ell+1}), \pi(\ell+1))$ for $\ell+1 \leq |\mathcal{S}|$. Here, we note that $\pi(\overline{\ell+1}) = \pi(\ell+1)$ if $\Delta_{\pi(\ell+1)} = 1$ (i.e., if $v_{\pi(\ell+1)}$ is checkpointed). Otherwise, we have $\pi(\overline{\ell+1}) = \pi(\bar{\ell})$. Then, from Equation (10.9), we derive:

$$\begin{aligned}\mathbb{E}_\ell &= \mathbb{E}_{\ell+1} + \big(\alpha\left(R_{\ell+1} + T_{\ell+1} + C_{\ell+1}\right) + \beta R_{\ell+1} + \gamma\big) \cdot \mathbb{P}\left(Y > t_\ell\right) \\ &\quad + \beta\big((1-\delta_{\ell+1})T_{\ell+1} + C_{\ell+1}\big) \cdot \mathbb{P}\left(Y > t_{\ell+1}\right) \\ &\geq \mathbb{E}_{\text{ckpt}}(\pi(\overline{\ell+1}), \pi(\ell+1)) \\ &\quad + \Big(\alpha\left(R_{\ell+1} + (t_{\ell+1} - t_{\bar{\ell}}) + C_{\ell+1}\right) + \beta R_{\ell+1} + \gamma\Big) \cdot \mathbb{P}\left(Y > v_{\pi(\ell)}\right) \\ &\quad + \beta\big((1-\delta_{\ell+1})(t_{\ell+1} - t_{\bar{\ell}}) + C_{\ell+1}\big) \cdot \mathbb{P}\left(Y > v_{\pi(\ell+1)}\right) \\ &= \mathbb{E}_{\text{ckpt}}\left((1-\Delta_{\pi(\ell+1)})\pi(\bar{\ell}) + \Delta_{\pi(\ell+1)}\pi(\ell+1), \pi(\ell+1)\right) \\ &\quad + \Big(\alpha\big(\mathbb{1}_{\pi(\bar{\ell})\neq 0}R + (v_{\pi(\ell+1)} - v_{\pi(\bar{\ell})}) + \Delta_{\pi(\ell+1)}C\big) + \beta \mathbb{1}_{\pi(\bar{\ell})\neq 0}R + \gamma\Big) \cdot \sum_{k=\pi(\ell)+1}^{n} f_k \\ &\quad + \beta\left((1-\Delta_{\pi(\ell+1)})(v_{\pi(\ell+1)} - v_{\pi(\bar{\ell})}) + \Delta_{\pi(\ell+1)}C\right) \cdot \sum_{k=\pi(\ell+1)+1}^{n} f_k \\ &\geq \mathbb{E}_{\text{ckpt}}\left(\pi(\bar{\ell}), \pi(\ell)\right)\end{aligned}$$

In the derivation above, the first inequality is due to the inductive hypothesis, and using $T_{\ell+1} = t_{\ell+1} - t_{\bar{\ell}}$ from Equation (9.7). The last inequality is due to the definition of $\mathbb{E}_{\text{ckpt}}$. Thus, by induction, we get $\mathbb{E}_{\text{ckpt}}(0,0) = \mathbb{E}_{\text{ckpt}}(\pi(\bar{0}), \pi(0)) \leq \mathbb{E}_0 = \mathbb{E}(\mathcal{S})$. This shows that $\mathbb{E}_{\text{ckpt}}(0,0)$ is not greater than the expected cost of any reservation sequence $\mathcal{S}$, thus returning the optimal solution.

Finally, one can pre-compute values of $\sum_{k=\ell+1}^{n} f_k$ for all $0 \leq \ell < n$ (in linear time) and store them. Then, computing $\mathbb{E}_{\text{ckpt}}(i_c, i_l)$ depends on $2(n - i_l)$ other $\mathbb{E}_{\text{ckpt}}$ values, thus takes $O(n - i_l)$ time. The overall complexity is therefore $O(n^3)$. $\square$

10.1.4 Other heuristics for STOCHASTIC-CKPT problem

All the results presented in Sections 10.1.1 to 10.1.3, namely the cost model (Theorem 4), the approximation algorithm for continuous distributions with bounded support (Theorem 5) and the optimal algorithm for discrete distributions (Theorem 6) can be extended to some variants of the problem where the checkpoint strategy is determined a priori.

A heuristic that checkpoints every reservation

Indeed, there are two important and natural variants to consider: strategies where no reservation is checkpointed, and strategies where all reservations are checkpointed. The former variant (called NO-CHECKPOINT) is presented later on in Section 10.2.4, where we derive an optimal algorithm for discrete distributions with reduced time complexity $O(n^2)$ instead of $O(n^3)$ as in Theorem 6. The latter variant (called ALL-CHECKPOINT) also admits an optimal dynamic programming algorithm of reduced time complexity $O(n^2)$:

Theorem 7. *For a discrete distribution $Y \sim (v_i, f_i)_{1 \leq i \leq n}$, the optimal expected cost for* ALL-CHECKPOINT *(when all reservations are checkpointed) is returned by $\mathbb{E}_{\text{AllCkpt}}(0)$, where $v_0 = 0$ and:*

$$\mathbb{E}_{\text{AllCkpt}}(n) = \beta \cdot \mathbb{E}[Y]$$

$$\mathbb{E}_{\text{AllCkpt}}(i) = \min_{i+1 \leq j \leq n} \left(\mathbb{E}_{\text{AllCkpt}}(j) + \beta C \cdot \sum_{k=j+1}^{n} f_k + \Big(\alpha \big(\mathbb{1}_{i \neq 0} R + (v_j - v_i) + \mathbb{1}_{j \neq n} C \big) + \beta \mathbb{1}_{i \neq 0} R + \gamma \Big) \cdot \sum_{k=i+1}^{n} f_k \right)$$

The optimal solution can be computed in $O(n^2)$ time.

While this heuristic requires the distribution of the execution time to be discrete, one can still use it for any continuous distribution by following discretization procedures further introduced in Section 10.2.4 for algorithmic developments for the application model without checkpoints.

A periodic heuristic

In addition to the algorithms presented in Section 10.1, we propose a periodic heuristic for the case of bounded distributions. This strategy, described in Algorithm 5, is a natural policy, where successive reservations differ in length by a constant amount of time T, called the *period*. A checkpoint is performed at the end of each period. Hence, the value of W_i associated with each t_i is constant in this strategy. The algorithm specifies the number of chunks τ in the domain $[a, b]$ of the bounded distribution, thus the period can be computed as $T = \frac{b-a}{\tau}$.

Algorithm 5 ALL-CHECKPOINT-PERIODIC(X, τ)

1: Let $[a, b]$ be the domain of X, and let $T = \frac{b-a}{\tau}$

2: $(t_i, \delta_i) = \begin{cases} (a + i \cdot T, 1) & \text{for } i = 1, 2, \ldots, \tau - 1 \\ (b, 0) & \text{for } i = \tau \end{cases}$

3: **return** $\mathcal{S}_\tau^{\text{period}} \leftarrow ((t_i, \delta_i))_{1 \leq i \leq \tau}$

We also define NO-CHECKPOINT-PERIODIC as the version of ALL-CHECKPOINT-PERIODIC where no checkpoint is performed. It is presented by Algorithm 6.

Algorithm 6 NO-CHECKPOINT-PERIODIC(X, τ)

1: Let $[a, b]$ be the domain of X, and let $T = \frac{b-a}{\tau}$

2: $(t_i, \delta_i) = \begin{cases} (a + i \cdot T, 0) & \text{for } i = 1, 2, \ldots, \tau - 1 \\ (b, 0) & \text{for } i = \tau \end{cases}$

3: **return** $\mathcal{S}_\tau^{\text{period}} \leftarrow ((t_i, \delta_i))_{1 \leq i \leq \tau}$

For such solutions, one can derive optimal strategies for specific distributions. We focus now on ALL-CHECKPOINT version of periodic strategies and prove this assertion for Uniform distributions. One can also prove that ALL-CHECKPOINT and its periodic counterpart are identical. The next subsections are dedicated to proving those assertions.

Periodic heuristic for exponential distributions

In this section, we are interested in proving that ALL-CHECKPOINT and ALL-CHECKPOINT-PERIODIC are similar for Exponential distribution.

Theorem 8. *When the execution time of a job follows a distribution $\mathcal{D} \sim$ Exponential (λ) , the solution of* ALL-CHECKPOINT *and* ALL-CHECKPOINT-PERIODIC *are identical.*

Proof. For this specific result, we express the solutions under the form: $\{T_1, T_2, T_3, \cdots\}$.

We have the following properties:

- Given $(t_1, t_2, \ldots,)$ a solution to ALL-CHECKPOINT, then the associated $\{T_1, T_2, T_3, \cdots\}$ satisfy:
$$\forall i, t_i = \sum_{j \leq i} T_j$$
This is a direct corollary of Equations (9.3) and (9.4).

- For $\mathcal{D} \sim$ Exponential (λ),
$$\mathbb{P}\left(X > V_1 + V_2\right) = \mathbb{P}\left(X > V_1\right) \cdot \mathbb{P}\left(X > V_2\right). \tag{10.11}$$

We define $(u_1, u_2, \cdots)$ the solution that minimizes

$$\sum_{i=1}^{\infty} \Big(\alpha\big(R + u_i + C\big) + \beta\big(R + C\big) + \gamma\Big) \cdot \mathbb{P}\left(X > \sum_{j<i} u_j\right) \tag{10.12}$$

To study $\mathcal{S}^{\text{opt}} = (T_1^o, T_2^o, \ldots)$ the optimal solution given by ALL-CHECKPOINT for $\mathcal{D}$, we rewrite the expected cost of a solution $\mathcal{S} = (T_1, T_2, \ldots)$ for ALL-CHECKPOINT:

$$\begin{aligned}
\mathbb{E}(\mathcal{S}) &= \beta \cdot \mathbb{E}[X] + \sum_{i=1}^{\infty} \Big(\alpha W_i + \beta\big(R_i + (1 - \delta_{i-1})T_{i-1} + C_{i-1}\big) + \gamma\Big) \cdot \mathbb{P}\left(X > t_{i-1}\right) \\
&= \beta \cdot \mathbb{E}[X] - C + \sum_{i=1}^{\infty} \Big[\alpha\big(R + T_i + C\big) + \beta\big(R + C\big) + \gamma\Big] \cdot \mathbb{P}\left(X > \sum_{j<i} T_j\right)
\end{aligned}$$

From this formulation, we can see that $T_1^o = u_1$, $T_2^o = u_2$, $T_3^o = u_3$, $\ldots$ as they satisfy the same equations.

By rewriting and using Equation (10.11),

$$\begin{aligned}
\mathbb{E}(\mathcal{S}) &= \beta \cdot \mathbb{E}[X] + \left(\alpha W_1 + \beta R + \gamma\right) + \sum_{i=2}^{\infty} \Big(\alpha W_i + \beta\big(R + C\big) + \gamma\Big) \cdot \mathbb{P}\left(X > \sum_{j<i} T_j\right) \\
&= \beta \cdot \mathbb{E}[X] + \alpha\big(W_1 + \beta C + \gamma\big) \\
&\qquad + \mathbb{P}\left(X > T_1\right) \cdot \sum_{i=2}^{\infty} \Big(\alpha\big(R + T_i + C\big) + \beta\big(R + C\big) + \gamma\Big) \cdot \mathbb{P}\left(X > \sum_{2 \leq j<i} T_j\right) \\
&= \beta \cdot \mathbb{E}[X] + \alpha\big(W_1 + \beta C + \gamma\big) \\
&\qquad + \mathbb{P}\left(X > T_1\right) \cdot \sum_{i=1}^{\infty} \Big(\alpha\big(R + T_{i-1} + C\big) + \beta\big(R + C\big) + \gamma\Big) \cdot \mathbb{P}\left(X > \sum_{j<i-1} T_{j+1}\right)
\end{aligned} \tag{10.13}$$

From this last formulation, we can see that the sequence $(T_2^o, T_3^o, \dots)$ that minimizes Equation (10.13) can be optimized independently of T_1^o, and that it is the solution that minimizes Equation (10.12). Hence we obtain, $T_2^o = u_1$, $T_3^o = u_2$,…Iterating the process, we obtain the result: $T_1^o = u_1 = T_2^o = u_2 = T_3^o = \dots$, and the solution to ALL-CHECKPOINT is a periodic solution.

□

Periodic heuristic for Uniform distributions

In this section, we consider the ALL-CHECKPOINT approach for Uniform distributions in the RESERVATIONONLY scenario. Specifically, we are able to characterize the best periodic approach: for a Uniform distribution $\mathcal{D}$ over $[a, b]$ (where $0 \leq a < b$), consider a reservation sequence $\mathcal{S}^{(n)} = (W_i, 1)_{1 \leq i \leq n}$ where $n \geq 2$, $W_1 = a + \frac{b-a}{n} + C$, $W_i = R + \frac{b-a}{n} + C$ for $1 < i < n$ and $W_n = R + \frac{b-a}{n}$. In other words, we have $n-1$ evenly distributed checkpoints in the interval $[a, b]$. The first reservation has a checkpoint but no restart, the last reservation (whenever used) has a restart but no checkpoint, and all intermediate reservations have both a restart and a checkpoint. Finally, let $\mathcal{S}^{(1)} = (b, 0)$ be the sequence with a unique reservation of length (and cost) b (no need to checkpoint in this particular case). The following proposition provides the optimal value of n:

Proposition 3. *With the above notations, $\mathbb{E}(\mathcal{S}^{(n)})$ is minimized either for $n = \max(1, \lfloor n^{\text{opt}} \rfloor)$ or $n = \lceil n^{\text{opt}} \rceil$ where $n^{\text{opt}} = \sqrt{\frac{b-a-2C}{C+R}}$ if $b - a \geq 2C$ and $n^{\text{opt}} = 1$ otherwise.*

Proof. Because the distribution is uniform, the probability that i reservations are needed for a given job is always equal to $\frac{1}{n}$, hence the expected cost of $\mathcal{S}^{(n)}$ for $n \geq 2$ is

$$\mathbb{E}(\mathcal{S}^{(n)}) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} W_j$$

where W_i is the cost of the i-th reservation. After several algebraic manipulations, we derive that

$$\mathbb{E}(\mathcal{S}^{(n)}) = \frac{n-1}{2n} a + \frac{n+1}{2n} b + \frac{n^2+n-2}{2n} C + \frac{n-1}{2} R$$

Differentiating, the derivative gets zeroed for $n = n^{\text{opt}}$ when $b - a \geq 2C$, and otherwise it stays positive, hence the result. □

If $\mathcal{D} \sim \text{Uniform}(a, b)$ with $[a, b] = [2, 20]$ and $C = R = 1$ we find $n^{\text{opt}} = \sqrt{8}$. One can compute that the cost for three reservations (ceil value for $\sqrt{8}$) is $\mathbb{E}(\mathcal{S}^{(3)}) = 97/6 \approx 16.2$. Let us now define a two-reservation strategies at milestones 11 and 20 [1], we can compute

[1] The first reservation will be of length $11 + 1$ with the checkpointing overhead, and the second one of length $1 + 9$ with the restart cost.

its associated cost by:

$$\mathbb{E}(\mathcal{S}^{(2)}) = \int_2^{11} (\alpha 12 + \beta t + \gamma)\mathbb{P}(X = t | X \leq 11) dt$$
$$+ \int_{11}^{20} [(\alpha 12 + \beta 12 + \gamma) + (\alpha 10 + \beta(t - 11) + \gamma)]\mathbb{P}(X = t | X \geq 11) dt$$

This gives us $\mathbb{E}(\mathcal{S}^{(2)}) = 17$ for distribution $\mathcal{D}$. Hence, it is more efficient to use three reservations than two.

Summary of contributions

To summarize the solutions for STOCHASTIC-CKPT problem described in this section, we provided:

- a $1 + \varepsilon$-approximation algorithm when execution time follows a bounded continuous distribution (Theorem 5),
- an optimal dynamic programming algorithm when execution time follows a discrete distribution (Theorem 6),
- a periodic heuristic and a variant of Theorem 6 where all reservations are checkpointed. We also derive properties of these heuristics for Exponential and Uniform distributions.

In the next section, we propose to study the problem when checkpoint/restart is not possible for the application.

10.2 Solutions without checkpointing

It is sometimes not possible to use checkpointing for a considered application. We briefly here discuss the main reasons of this assertion. More details are presented in Chapter 8. There are two main paradigm for managing checkpointing within an application. Checkpointing may be performed either by the application itself by explicitly modifying the code in order to work with a user level checkpoint library (like FTI [27]), or by linking an external library. The first case requires modifications in the code of the application, which is often not suitable for stochastic applications due to the dynamic nature of their code. The second solution seems to be the most suitable one. However, we already mentioned that some solutions are no longer maintained (BLCR [70]) and often do not support checkpointing for containers (DMTCP [19]). It is an important reason, as stochastic applications emerge from ML/Big-Data frameworks that are originally designed for cloud computing, where developing an application inside a container is the *de facto* approach. Even though one could checkpoint the application inside the container without checkpointing the whole container, we consider that such solution requires an important human cost to be implemented. Hence, this is not a suitable solution for some users that are not native developers.

The contributions of this section are as follows:

- A refinement of STOCHASTIC-CKPT without checkpointing considerations, with all necessary formulations (Section 10.2.1)
- A characterization of an optimal solution and a recursive formula to compute a sequence of reservation in the case of continuous distribution, up to the computation of the first reservation (Section 10.2.2)
- A heuristic to derive a sub-optimal solution using an exhaustive search procedure to estimate the value of the first reservation for any continuous distribution (Section 10.2.3)
- An optimal solution when the walltime of application follows a discrete probability distribution (Section 10.2.4)
- A discretization-based heuristic to use the discrete dynamic-programming algorithm with any continuous distribution (Section 10.2.4)
- A discussion on the extension of the results presented in this section with non-convex cost functions (Section 10.2.5)

We start this study by presenting the formulation of the problem without checkpointing.

10.2.1 Problem refinement

Similarly to the case with checkpoint, the user makes a series of reservations, until the job successfully executes within the length of the last reservation. If a reservation ends without the success of the application, the work done during this reservation is lost and one has to restart the application from scratch. This fact shows the benefits of using checkpointing, when it is supported by the considered application.

For now, we consider that checkpointing is no more available. For a sequence $S = (W_1, W_2, \dots, W_i, W_{i+1}, \dots)$, and for a job execution time t, the cost is

$$C(k,t) = \sum_{i=1}^{k-1} (\alpha W_i + \beta W_i + \gamma) + \alpha W_k + \beta t + \gamma \tag{10.14}$$

where k is the smallest index in the sequence such that $t \leq W_k$ (or equivalently, $W_{k-1} < t \leq W_k$; recall that $W_0 = 0$).

The goal is to find a scheduling strategy, i.e., a sequence of increasing reservation durations, that minimizes the cost in expectation. Formally, the expected cost for a sequence $S = (W_1, W_2, \dots, W_i, W_{i+1}, \dots)$ can be written as:

$$\mathbb{E}(S) = \sum_{k=1}^{\infty} \int_{W_{k-1}}^{W_k} C(k,t) f(t) dt \tag{10.15}$$

Indeed, when $W_{k-1} < t \leq W_k$, the cost is $C(k,t)$, which we weight with the corresponding probability.
Here are two examples for two usual continuous distributions:

- UNIFORM(a, b): for a uniform distribution over the interval $[a, b]$ where $0 < a < b$, we have $f(t) = \frac{1}{b-a}$ if $a \leq t \leq b$, and $f(t) = 0$ otherwise. Given a finite sequence $S = (\frac{a+b}{2}, b)$, the expected cost is

$$\begin{aligned}\mathbb{E}(S) &= \int_a^{\frac{a+b}{2}} (\alpha\frac{a+b}{2} + \beta t + \gamma)\frac{1}{b-a}dt \\ &+ \int_{\frac{a+b}{2}}^{b} \big((\alpha\frac{a+b}{2} + \beta\frac{a+b}{2} + \gamma) + (\alpha b + \beta t + \gamma)\big)\frac{1}{b-a}dt\end{aligned}$$

  The first term is for values of t that are in $[a, \frac{a+b}{2}]$ and the second term is for larger values of t in $[\frac{a+b}{2}, b]$. For the latter term, we pay a constant cost

$$\alpha\frac{a+b}{2} + \beta\frac{a+b}{2} + \gamma$$

  for the first reservation, which was unsuccessful, and then a cost that depends upon the value of t for the second reservation if $\beta \neq 0$.

- EXP(λ): for an exponential distribution with rate λ and support in $[0, \infty)$, we have $f(t) = \lambda e^{-\lambda t}$ for all $t \geq 0$. Given an infinite and unbounded sequence $S = (\frac{1}{\lambda}, \frac{2}{\lambda}, \dots, \frac{i}{\lambda}, \frac{i+1}{\lambda}, \dots)$, the expected cost is

$$\mathbb{E}(S) = \sum_{k=1}^{\infty} \int_{\frac{k-1}{\lambda}}^{\frac{k}{\lambda}} \Big(\sum_{i=1}^{k-1}(\alpha\frac{i}{\lambda} + \beta\frac{i}{\lambda} + \gamma) + \alpha\frac{k}{\lambda} + \beta t + \gamma\Big)\lambda e^{-\lambda t}dt$$

  Again, when $t \in [\frac{k-1}{\lambda}, \frac{k}{\lambda}]$, we pay a fixed cost for the $k-1$ first reservations, and a possibly variable cost for the k-th reservation. Looking at the expression of $\mathbb{E}(S)$ above, we easily see that the given sequence S has a finite expected cost $\mathbb{E}(S)$. In fact, there are many sequences with finite expected cost, such as those defined by $t_i = ui + v$ for $i \geq 1$, where u and v are positive constants.

We are now ready to state the optimization problem:

Definition 4 (STOCHASTIC)**.** Given a probability distribution (with CDF F) for the execution times of stochastic jobs, and given a cost function given by Equation (9.1) (with parameters α, β and γ), find a reservation sequence S with minimal expected cost $\mathbb{E}(S)$ as given in Equation (10.15).

As we stated before, we focus on the usual probability distributions, hence we assume that the density function f and the CDF F of $\mathcal{D}$ are smooth (infinitely differentiable), and that $\mathcal{D}$ has finite expectation.

10.2.2 Execution time as a continuous probability distribution

In this section, we establish key properties of an optimal solution for a continuous probability distribution of execution time.

A new expression for cost function

We start by establishing a simpler expression for the cost function of STOCHASTIC.

Theorem 9. *Given a sequence $S = (t_1, t_2, \ldots, t_i, t_{i+1}, \ldots)$, the cost function given by Equation (10.15) (with parameters α, β and γ) can be rewritten as (with $t_0 = 0$):*

$$\mathbb{E}(S) = \beta \cdot \mathbb{E}[X] + \sum_{i=0}^{\infty} (\alpha t_{i+1} + \beta t_i + \gamma)\, \mathbb{P}\left(X \geq t_i\right) \tag{10.16}$$

Proof. We first expand Equation (10.15) as follows:

$$\mathbb{E}(S) = \sum_{k=1}^{\infty} \left(\int_{t_{k-1}}^{t_k} \Big(\sum_{i=1}^{k} (\alpha t_i + \gamma) + \sum_{i=1}^{k-1} \beta t_i + \beta t \Big) f(t) dt \right) \tag{10.17}$$

We compute the three terms on the right-hand side separately. By defining $t_0 = 0$, the first term can be expressed as:

$$\begin{aligned} \sum_{k=1}^{\infty} \left(\int_{t_{k-1}}^{t_k} \Big(\sum_{i=1}^{k} (\alpha t_i + \gamma) \Big) f(t) dt \right) &= \sum_{k=1}^{\infty} \sum_{i=1}^{k} (\alpha t_i + \gamma) \int_{t_{k-1}}^{t_k} f(t) dt \\ &= \sum_{k=1}^{\infty} \sum_{i=1}^{k} (\alpha t_i + \gamma)\, \mathbb{P}\left(X \in [t_{k-1}, t_k]\right) \\ &= \sum_{i=1}^{\infty} \sum_{k=i}^{\infty} (\alpha t_i + \gamma)\, \mathbb{P}\left(X \in [t_{k-1}, t_k]\right) \\ &= \sum_{i=1}^{\infty} (\alpha t_i + \gamma)\, \mathbb{P}\left(X \geq t_{i-1}\right) \end{aligned}$$

Similarly, we obtain the second term:

$$\sum_{k=1}^{\infty}\left(\int_{t_{k-1}}^{t_k}\Big(\sum_{i=1}^{k-1}\beta t_i\Big)f(t)dt\right)=\sum_{i=1}^{\infty}\beta t_i\mathbb{P}\left(X\geq t_i\right)$$

and the third term:

$$\sum_{k=1}^{\infty}\left(\int_{t_{k-1}}^{t_k}\beta t f(t)dt\right)=\beta\cdot\mathbb{E}[X]$$

Plugging these three terms back into Equation (10.17), we get the desired expression for the cost function as given by Equation (10.16). □

Characterizing the optimal solution

In this section, we are interested in characterizing the properties of an optimal solution $S=(t_1^o,t_2^o,\cdots,t_n^o,\cdots)$ for STOCHASTIC.

Upper bound on t_1^o and finite expected cost In this section, we extract an upper bound for the first request t_1^o of an optimal sequence S^o to STOCHASTIC, which allows us to show that the expected cost $\mathbb{E}(S^o)$ is upper bounded too, and hence finite. This result holds in a general setting, namely, for any distribution $\mathcal{D}$ such that $\mathbb{E}(X^2)<\infty$.

Obviously, if the distribution's support is upper bounded, such as for UNIFORM(a,b), a solution is to choose that upper bound for t_1^o (e.g., $t_1^o\leq b$ for UNIFORM(a,b)). Hence, we focus on distributions with infinite support $[a,\infty)$ and aim at restricting the search for an optimal t_1^o to a bounded interval $[a,A_1]$ for some A_1. We derive the following result.

Theorem 10. *For any distribution $\mathcal{D}$ with infinite support $[a,\infty)$ such that $\mathbb{E}[X^2]<\infty$, the value t_1^o of an optimal sequence $S^o=(t_1^o,t_2^o,\dots,t_i^o,t_{i+1}^o,\dots)$ satisfies $t_1^o\leq A_1$, and $\mathbb{E}(S^o)\leq A_2$, where*

$$A_1=\mathbb{E}[X]+1+\frac{\alpha+\beta}{2\alpha}(\mathbb{E}[X^2]-a^2)+\frac{\alpha+\beta+\gamma}{\alpha}(\mathbb{E}[X]-a) \tag{10.18}$$

$$A_2=\beta\cdot\mathbb{E}(X)+\alpha A_1+\gamma \tag{10.19}$$

Proof. We consider the sequence $S=(t_1,t_2,\dots,t_i,t_{i+1},\dots)$ with $t_i=a+i$ for $i\geq 1$ (and

$t_0 = 0$), and compute:

$$\begin{aligned}
\mathbb{E}(S) - \beta \cdot \mathbb{E}[X] &= \sum_{i=0}^{\infty} (\alpha t_{i+1} + \beta t_i + \gamma) \mathbb{P}\left(X \geq t_i\right) \\
&= \sum_{i=0}^{\infty} (\alpha(a+i+1) + \beta(a+i) + \gamma) \mathbb{P}\left(X \geq a+i\right) \\
&= \alpha(a+1) + \gamma + \sum_{i=1}^{\infty} (\alpha+\beta)(a+i) \mathbb{P}\left(X \geq a+i\right) \\
&\quad + (\alpha+\gamma) \sum_{i=1}^{\infty} \mathbb{P}\left(X \geq a+i\right) \\
&= \ \alpha(a+1) + \gamma + (\alpha+\beta) \sum_{i=1}^{\infty} \int_{a+i-1}^{a+i} (a+i) \mathbb{P}\left(X \geq a+i\right) dt \\
&\qquad + (\alpha+\gamma) \sum_{i=1}^{\infty} \int_{a+i-1}^{a+i} \mathbb{P}\left(X \geq a+i\right) dt
\end{aligned}$$

Note that for all $t \in [a+i-1, a+i]$, we have both

$$a + i \leq t + 1$$

and

$$\mathbb{P}\left(X \geq a+i\right) \leq \mathbb{P}\left(X \geq t\right)$$

Thus,

$$(a+i)\mathbb{P}\left(X \geq a+i\right) \leq (t+1)\mathbb{P}\left(X \geq t\right)$$

Hence, we can write:

$$\begin{aligned}
\mathbb{E}(S) - \beta \cdot \mathbb{E}[X] \leq \ & \alpha(a+1) + \gamma + (\alpha+\beta) \sum_{i=1}^{\infty} \int_{a+i-1}^{a+i} (t+1) \mathbb{P}\left(X \geq t\right) dt \\
& + (\alpha+\gamma) \sum_{i=1}^{\infty} \int_{a+i-1}^{a+i} \mathbb{P}\left(X \geq t\right) dt \\
= \ & \alpha(a+1) + \gamma + (\alpha+\beta) \int_a^{\infty} (t+1) \mathbb{P}\left(X \geq t\right) dt + (\alpha+\gamma) \int_a^{\infty} \mathbb{P}\left(X \geq t\right) dt \\
\leq \ & \alpha(a+1) + \gamma + (\alpha+\beta) \int_a^{\infty} t \cdot \mathbb{P}\left(X \geq t\right) dt + (2\alpha+\beta+\gamma) \int_a^{\infty} \mathbb{P}\left(X \geq t\right) dt
\end{aligned}$$

For the last inequality, we have split

$$\int_a^{\infty} (t+1) \mathbb{P}\left(X \geq t\right) dt$$

into
$$\int_a^\infty t\mathbb{P}\left(X \geq t\right) dt$$
and
$$\int_a^\infty \mathbb{P}\left(X \geq t\right) dt$$

Extending the support of $\mathcal{D}$ to $[0, \infty)$ by letting $f(t) = 0$ for $0 \leq t \leq a$, and hence $\mathbb{P}\left(X \geq t\right) = 1$ for $0 \leq t \leq a$, we have the following property for any integer $p \geq 1$:

$$\begin{aligned}
\int_0^\infty t^{p-1} \cdot \mathbb{P}\left(X \geq t\right) dt &= \int_{t=0}^\infty t^{p-1} \int_{x=t}^\infty f(x) dx dt \\
&= \int_{x=0}^\infty f(x) \int_{t=0}^x t^{p-1} dt dx \\
&= \int_0^\infty \frac{x^p}{p} f(x) dx \\
&= \frac{\mathbb{E}[X^p]}{p}
\end{aligned}$$

Hence, using $p = 1$, we have:

$$\int_a^\infty \mathbb{P}\left(X \geq t\right) dt = \int_0^\infty \mathbb{P}\left(X \geq t\right) dt - \int_0^a \mathbb{P}\left(X \geq t\right) dt = \mathbb{E}[X] - a$$

and using $p = 2$, we get:

$$\int_a^\infty t \cdot \mathbb{P}\left(X \geq t\right) dt = \int_0^\infty t \cdot \mathbb{P}\left(X \geq t\right) dt - \int_0^a t \cdot \mathbb{P}\left(X \geq t\right) dt = \frac{\mathbb{E}[X^2] - a^2}{2}$$

Altogether, we derive that:

$$\mathbb{E}(S) \leq \beta \cdot \mathbb{E}[X] + \alpha A_1 + \gamma \tag{10.20}$$

where A_1 is given by Equation (10.18). From Equation (10.16), the expected cost of any sequence S satisfies $\mathbb{E}(S) \geq \beta \cdot \mathbb{E}[X] + \alpha t_1 + \gamma$ (cost of expected execution time and cost of first request). Hence, necessarily in an optimal sequence, the first reservation t_1^o satisfies to $t_1^o \leq A_1$. Thus, Equation (10.18) gives the desired bound on t_1^o. □

Properties of optimal sequences We now derive a recurrence relation between the successive requests in the optimal sequence for STOCHASTIC.

Theorem 11. *Let $S^o = (t_i^o)_{i \geq 1}$ denote an optimal sequence for* STOCHASTIC. *For all $i \geq 1$, if t_i^o is not the last element of the sequence and $F(t_i^o) \neq 1$, we have the following property:*

$$\alpha t_{i+1}^o + \beta t_i^o + \gamma = \alpha \frac{1 - F(t_{i-1}^o)}{f(t_i^o)} + \beta \frac{1 - F(t_i^o)}{f(t_i^o)} \tag{10.21}$$

Proof. We fix an index $j \geq 1$ such that $F(t_j^o) \neq 1$ and consider the expected cost when we replace t_j^o by an arbitrary value $t \in [t_{j-1}^o, t_{j+1}^o]$. This amounts to using the sequence

$$S_j^o(t) = (t_1^o, t_2^o, \cdots, t_{j-1}^o, t, t_{j+1}^o, \cdots)$$

whose expected cost, according to Equation (10.16), is the following:

$$\mathbb{E}(S_j^o(t)) = \beta \cdot \mathbb{E}[X] + \sum_{i \neq j-1, j} (\alpha t_{i+1}^o + \beta t_i^o + \gamma)\mathbb{P}(X \geq t_i^o) + (\alpha t + \beta t_{j-1}^o + \gamma)\mathbb{P}\left(X \geq t_{j-1}^o\right)$$
$$+ (\alpha t_{j+1}^o + \beta t + \gamma)\mathbb{P}(X \geq t)$$

which we can rewrite as:

$$\mathbb{E}(S_j^o(t)) = C_j + \alpha t(1 - F(t_{j-1}^o)) + (\alpha t_{j+1}^o + \beta t + \gamma)(1 - F(t))$$

where C_j is some constant independent of t. By definition, the minimum of $\mathbb{E}(S_j^o(t))$ on $[t_{j-1}^o, t_{j+1}^o]$ is achieved at $t = t_j^o$ (and potentially at other values). Because $\mathbb{E}(S_j^o(t))$ is smooth, we have that: its derivative at t_j^o, which is not an extremity of the interval $[t_{j-1}^o, t_{i+1}^o]$, must be equal to zero, i.e., $\frac{\partial \mathbb{E}(S_j^o(t))}{\partial t} = 0$. This gives:

$$\alpha(1 - F(t_{j-1}^o)) + \beta(1 - F(t_j^o)) - (\alpha t_{j+1}^o + \beta t_j^o + \gamma) f(t_j^o) = 0 \tag{10.22}$$

To get the final result, it remains to show that $f(t_j^o) \neq 0$. Otherwise, we would get from Equation (10.22) that:

$$\alpha(1 - F(t_{j-1}^o)) + \beta(1 - F(t_j^o)) = 0$$

which implies that:

$$F(t_{j-1}^o) = 1$$

because $\alpha > 0$ (and $\beta(1 - F(t_j^o)) \geq 0$). But then,

$$F(t_j^o) \geq F(t_{j-1}^o) = 1$$

which contradicts the initial assumption. Hence, $f(t_j^o) \neq 0$, and rewriting Equation (10.22) directly leads to Equation (10.21). □

Note that the condition $F(t_i^o) \neq 1$ in Theorem 11 applies to distributions with bounded support, such as UNIFORM(a, b), where $F(b) = 1$.
For the usual distributions with unbounded support, such as EXP(λ), we have $F(t) < 1$ for all $t \in [0, \infty)$ and an optimal sequence must be infinite. In essence, Theorem 11 suggests that an optimal sequence is characterized solely by its first value t_1^o:

Proposition 4. *For a smooth distribution with unbounded support, solving* STOCHASTIC *reduces to finding t_1^o that minimizes*

$$\sum_{i=0}^{\infty} (\alpha t_{i+1} + \beta t_i + \gamma)\, \mathbb{P}(X \geq t_i)$$

where $t_0^o = 0$, and for all $i \geq 2$,

$$t_i^o = \frac{1 - F(t_{i-2}^o)}{f(t_{i-1}^o)} + \frac{\beta}{\alpha}\left(\frac{1 - F(t_{i-1}^o)}{f(t_{i-1}^o)} - t_{i-1}^o\right) - \frac{\gamma}{\alpha} \tag{10.23}$$

For a smooth distribution with bounded support, the recurrence in Equation (10.23) still holds but the optimal sequence stops as soon as it reaches t_i^o with $F(t_i^o) = 1$.

Proposition 4 provides an optimal algorithm for general smooth distributions, up to the determination of t_1^o. However, computing the optimal t_1^o, remains a difficult problem, except for simple distributions such as UNIFORM(a, b) (see Section 10.2.2).

Optimal solution for specific distributions

Uniform distributions In this section, we discuss the optimal strategy for a uniform distribution UNIFORM(a, b), where $0 < a < b$. Intuitively, one could try and make a first reservation of duration, say, $t_1 = \frac{a+b}{2}$, and then a second reservation of duration $t_2 = b$. However, we show that the best approach is to make a single reservation of duration $t_1 = b$, for any value of the parameters α, β and γ:

Theorem 12. *For a uniform distribution* UNIFORM(a, b)*, the optimal sequence for* STOCHASTIC *is $S^o = (b)$.*

Proof. We proceed by contradiction and assume there is an optimal sequence $S = (t_1, t_2, \dots, t_i, t_{i+1}, \dots)$ for STOCHASTIC where $t_1 < b$. Necessarily, this sequence contains more elements: either it is finite of length n and then necessarily $t_n = b$ (hence $n \geq 2$): otherwise $t_n < b$ and $\mathbb{E}(S) = \infty$ because the interval $[t_n, b]$ has non-zero measure; or it is infinite and then the conclusion holds (note that in that case, $\lim_{i\to\infty} t_i = b$: otherwise the strictly increasing sequence $(t_i)_{i\geq 1}$ converges to some value $b' < b$ and $\mathbb{E}(S) = \infty$ because the interval $[b', b]$ has non-zero measure). Altogether, t_2 always exists and $t_1 < t_2 \leq b$.

We can compute $\mathbb{E}(S)$ by distinguishing whether the job execution time t satisfies: (i) $a \leq t \leq t_1$; or (ii) $t_1 \leq t \leq t_2$; or (iii) $t_2 \leq t \leq b$. Note that the last case (iii) may disappear if $t_2 = b$. We obtain:

$$\begin{aligned}\mathbb{E}(S) = \frac{t_1 - a}{b - a}(\alpha t_1 + \beta\frac{a + t_1}{2} + \gamma) &+ \frac{t_2 - t_1}{b - a}(\alpha t_1 + \beta t_1 + \gamma + \alpha t_2 + \beta\frac{t_1 + t_2}{2} + \gamma) \\ &+ \frac{b - t_2}{b - a}(\alpha t_1 + \beta t_1 + \gamma + \alpha t_2 + \beta t_1 + \gamma + Z)\end{aligned}$$

In the equation above, we have used the fact that:

$$\beta \int_a^{t_1} t\mathbb{P}(X = t | a \leq t \leq t_1)dt = \beta\frac{a + t_1}{2}$$

and similarly,

$$\beta \int_{t_1}^{t_2} t\mathbb{P}(X = t | t_1 \leq t \leq t_2) dt = \beta \frac{t_1 + t_2}{2}$$

Also, $\frac{b-t_2}{b-a} Z$ represents the expected cost of the third and following reservations for $t \in [t_2, b]$.

Now, we suppress t_1 in the optimal sequence S and get a new sequence

$$S' = (t_2, t_3, \dots, t_i, t_{i+1}, \dots)$$

We can compute its expected cost just as before:

$$\mathbb{E}(S') = \frac{t_2 - a}{b - a}(\alpha t_2 + \beta \frac{a + t_2}{2} + \gamma) + \frac{b - t_2}{b - a}(\alpha t_2 + \beta t_2 + \gamma + Z)$$

where Z has the same value as above, because only the beginning of the sequence has been modified. We can then derive that:

$$\mathbb{E}(S) - \mathbb{E}(S') = \frac{1}{b - a}(\alpha u + \beta v + \gamma w)$$

where $u = t_1(b - t_2) + a(t_2 - t_1) > 0$, $v = t_1(b - t_1) > 0$, and $w = b - t_1 > 0$. Hence $\mathbb{E}(S) > \mathbb{E}(S')$, and S was not an optimal sequence, the desired contradiction. $\square$

Exponential distributions In this section, we provide partial results for the RESERVATIONONLY problem ($\beta = \gamma = 0$ and $\alpha = 1$) with an exponential distribution EXP(λ). From Theorem 10 (and the example in Section 10.2.1), we know that there exist sequences of finite expected cost. We further characterize the optimal solution as follows:

Proposition 5. *Let $S_1 = (s_1, s_2, \dots, s_i, s_{i+1}, \dots)$ denote the optimal sequence for* RESERVATIONONLY *with $X_1 \sim$* EXP(1). *It is the sequence that minimizes*

$$\mathbb{E}(S_1) = s_1 + 1 + \sum_{i=1}^{\infty} e^{-s_i},$$

such that, $s_2 = e^{s_1}$, and for $i \geq 3$,

$$s_i = e^{s_{i-1} - s_{i-2}} \tag{10.24}$$

We denote by $E_1 = s_1 + 1 + \sum_{i=1}^{\infty} e^{-s_i}$. The optimal sequence for RESERVATIONONLY *for $X_\lambda \sim$* EXP(λ) *is the infinite sequence $S_\lambda = (t_1, t_2, \dots, t_i, t_{i+1}, \dots)$ such that $t_i = \frac{s_i}{\lambda}$ for $i \geq 1$. Its expected cost is $\mathbb{E}(S_\lambda) = \frac{1}{\lambda} E_1$.*

Proof. The results on S_1 (Equation (10.24) and E_1) follow directly from Proposition 4 and Equation (10.23).

Consider an EXP(λ) distribution: X_λ. From Equation (10.16), the expected cost of the optimal sequence S_λ is

$$\mathbb{E}(S_\lambda) = \sum_{i=0}^{\infty} t_{i+1} e^{-\lambda t_i}$$

where $t_0 = 0$, t_1 is unknown, and the value of t_i for $i \geq 2$ is given by Equation (10.23) as

$$t_i = \frac{e^{\lambda(t_{i-1}-t_{i-2})}}{\lambda}$$

for $i \geq 2$. We define $u_i = \lambda t_i$ for all $i \geq 0$, we derive that

$$\mathbb{E}(S_\lambda) = \frac{1}{\lambda}\sum_{i=0}^{\infty} u_{i+1} e^{-u_i}$$

with $u_i = e^{u_{i-1}-u_{i-2}}$ for all $i \geq 2$. Hence u_i is the sequence that minimizes

$$\mathbb{E}(S_\lambda) = \sum_{i=0}^{\infty} t_{i+1} e^{-\lambda t_i} = \frac{1}{\lambda}\sum_{i=0}^{\infty} u_{i+1} e^{-u_i} = \frac{1}{\lambda}\left(u_1 + 1 + \sum_{i=1}^{\infty} e^{-u_i}\right)$$

We can notice that the sequence $\mathcal{U} = (u_1, u_2, \cdots, u_i, \cdots)$ solves the same system of equations as S_1, hence S_1 is a valid solution for $\mathcal{U}$.

Hence the result. □

Again, the optimal sequence is fully characterized by the value of t_1 or s_1. Here, s_1 is independent of λ. In other words, the solution for EXP(1) is generic, and the solution for EXP(λ) for an arbitrary λ can be directly derived from it. Unfortunately, we do not know how to compute s_1 analytically. However, a brute-force search provides the value $s_1 \approx 0.74219$, which means that the first reservation for EXP(λ) should be approximately three quarters of the mean value $\frac{1}{\lambda}$ of the distribution, for any $\lambda > 0$.

10.2.3 A heuristic solution for arbitrary continuous distributions

The results of the previous section provide an optimal strategy for STOCHASTIC up to the determination of the optimal t_1^o, since Theorem 11 and Proposition 4 allow us to compute the subsequent t_i^o's. However, while we have derived an upper bound on t_1^o, we do not know how to compute its exact value for an arbitrary distribution.

We present here an exhaustive search procedure called BRUTE-FORCE that tries different values for the first reservation length t_1 in a sequence S, and then computes the subsequent values according to Equation (10.23). Algorithm 7 details the different steps of the heuristic.

Specifically, we try M different values of t_1 on the interval $[a, b]$, where a is the lower bound of the distribution and b is the upper bound if the distribution is finite. Otherwise, we set $b = A_1$, which is an upper bound on the optimal t_1^o as given in Equation (10.18). For each $m = 1, \ldots, M$, we generate a sequence that starts with

$$t_1 = a + m \cdot \frac{b-a}{M}$$

Given a sequence S, we evaluate its expected cost using Equation (10.16). We finally return the minimum expected cost found over all the M values of t_1. Note that some values of t_1 may not lead to any result, because the sequence computed based on it and using Equation (10.23) may not be strictly increasing. In this case, we simply ignore the sequence. The complexity of this heuristic is $\mathcal{O}(M)$.

Algorithm 7 BRUTE-FORCE (X, M)

Let $[a, b]$ be the domain of definition of X, with $0 \leq a < b$
Let $t_1^{up} = b$ if $b \neq \infty$, $t_1^{up} = A_1$ otherwise as given in Equation (10.18)
Let $V = \{a + m \cdot \frac{t_1^{up} - a}{M}\}$ with $m = 1, \ldots, M$, the set of candidate t_1
Let $S^b = \emptyset$ the variable to track current best solution
Let $\mathbb{E}(S^b) = 0$ the expectation of current best solution
for $m \in V$ **do**
    Let S^c be the sequence associated to $t_1 = m$ following Equation (10.23)
    Let $\mathbb{E}(S^c)$ be the expectation of the cost of S^c as in Equation (10.16)
    if S^c strictly increasing and $\mathbb{E}(S^c) < \mathbb{E}(S^b)$ **then**
        $S^b = S^c$, $\mathbb{E}(S^b) = \mathbb{E}(S^c)$
    end if
end for
return S^b

We point out that the actual optimal value for the first request t_1^o would possibly lie in between two successive values of t_1 that we try. However, because we deal with smooth probability distributions, we expect to return a t_1 and an associated expected cost that are close to the optimal when M is sufficiently large. In the performance evaluation, we set $M = 5000$.

10.2.4 Execution time as a discrete probability distribution

We focus in this section on solving STOCHASTIC-CKPT in the case of a discrete distribution of the application walltime.

An optimal algorithm for discrete distribution

When execution time of the application follows any discrete probability distribution, the optimal sequence is computed by a dynamic programming algorithm.

Theorem 13 (Discrete distribution). *If $X \sim (v_i, f_i)_{i=1\ldots n}$, then* STOCHASTIC *can be solved optimally in polynomial time. This algorithm is called* NO-CHECKPOINT.

Proof. Let $\mathbb{E}_i^*$ denote the optimal expected cost given that $X \geq v_i$. In this case, to compute the optimal expected cost, the probability distribution of X needs to be first updated as

$$f'_k = \frac{f_k}{\sum\limits_{j=i}^{n} f_j}, \forall k = i, \dots, n$$

which guarantees that $\sum\limits_{k=i}^{n} f'_k = 1$.

We can then express $\mathbb{E}^*_i$ based on the following dynamic programming formulation, NO-CHECKPOINT:

$$\mathbb{E}^*_i = \min_{i \le j \le n} \left(\alpha v_j + \gamma + \sum_{k=i}^{j} f'_k \cdot \beta v_k + \Big(\sum_{k=j+1}^{n} f'_k \Big) \big(\beta v_j + \mathbb{E}^*_{j+1} \big) \right)$$

In particular, to compute $\mathbb{E}^*_i$, we make a first reservation of all possible discrete values $(v_j)_{j=i \dots n}$ and select the one that incurs the minimum total expected cost. For each v_j considered, if the job's actual execution time is greater than v_j (with probability $\sum\limits_{k=j+1}^{n} f'_k$), the total cost also includes the optimal cost $\mathbb{E}^*_{j+1}$ for making subsequent reservations.

The dynamic program is initialized with $\mathbb{E}^*_n = \alpha v_n + \beta v_n + \gamma$, and the optimal total expected cost is given by $\mathbb{E}^*_1$. The complexity is $\mathcal{O}(n^2)$, since each $\mathbb{E}^*_i$ depends on $n - i$ other expected costs, with associated probability updates and summations that can be computed in $\mathcal{O}(n - i)$ time. The optimal sequence of reservations can be obtained by backtracking the decisions made at each step. □

Note that the sequence obtained by dynamic programming always ends with the largest value $v_n = b$. When applying it back to a continuous distribution with unbounded support, more values will be needed, because the sequence must tend to infinity as explained in Section 9.3. In this case, additional values can be appended to the sequence by using other heuristics, such as the ones presented next in Section 11.1.1.

Truncating and discretizing continuous distributions

We discuss in this section how one could use the above presented algorithm for any continuous distribution by discretizing it into a discrete distribution.

If a continuous distribution has finite support $[a, b]$, where $0 \le a < b$, then we can directly discretize it. Otherwise, for a distribution with infinite support $[a, \infty)$, where $0 \le a$, we need to first truncate it in order to operate on a bounded interval. In the latter case, we define

$$b = Q(1 - \epsilon)$$

where

$$Q(x) = \inf\{t | F(t) \ge x\}$$

is the quantile function. That is, we discard the final $\epsilon \in (0, 1)$ quantile of the distribution, which for usual distributions ensures that b is finite. In either case, the discretization will then be performed on the interval $[a, b]$. Let n denote the number of discrete values we will sample from the continuous distribution. The result will be a set of n pairs $(v_i, f_i)_{i=1\dots n}$, where the v_i's represent the possible execution times of the jobs, and the f_i's represent the corresponding probabilities. We envision two procedures for the discretization:

- EQUAL-PROBABILITY: This scheme ensures that all the discrete execution times have the same probability. Thus, for all $i = 1, 2, \dots, n$, we can compute

$$v_i = Q\big(i \cdot \frac{F(b)}{n}\big)$$

  with associated

$$f_i = \frac{F(b)}{n}$$

- EQUAL-TIME: This scheme makes the discrete execution times equally spaced in the interval $[a, b]$. Thus, for all $i = 1, 2, \dots, n$, the execution times and their probabilities are computed as

$$v_i = a + i \cdot \frac{b-a}{n}$$

  associated to

$$f_i = F(v_i) - F(v_{i-1})$$

  We show in next chapter that EQUAL-TIME scheme tends to perform better than EQUAL-PROBABILITY, hence we preferably use EQUAL-TIME procedure to transform continuous distributions into discrete ones when necessary.

Note that when the continuous distribution has unbounded support, the probabilities for the n discrete execution times do not sum up to 1, i.e.,

$$\sum_{i=1}^{n} f_i = F(b) = 1 - \epsilon$$

A smaller value of ϵ and a larger number n will provide a better sampling of the continuous distribution in either discretization scheme. In the performance evaluation, we set $\epsilon = 10^{-7}$ and $n = 1000$.

We saw in the previous section that our solution for STOCHASTIC for continuous distribution is a sub-optimal heuristic, hence not an exact solution. Thus, the idea of using the dynamic-programming algorithm by using discretization procedures comes naturally. We will compare in Chapter 11 the performance of the discretization-based heuristic with the heuristic solution for STOCHASTIC.

In Section 10.1.2, we presented a near-optimal solution for STOCHASTIC-CKPT using continuous distribution. One could also use the discretization procedures in that case, as we also provide an optimal solution in the case of discrete distribution. However, the FPTAS already generates a solution really close to the optimal. Hence, the interest of using the discretization is less prominent for STOCHASTIC-CKPT than for STOCHASTIC.

10.2.5 Extension to convex cost functions

We briefly show that the results presented in this section for the STOCHASTIC problem can be easily extended to convex cost functions.
To do so, we extend Theorem 11 and Proposition 4 in the case of a general convex cost function.

Theorem 14. *Let $S^o = (t_i^o)_{i\geq 1}$ denote an optimal sequence for* STOCHASTIC-CKPT. *For all $i \geq 1$, if t_i^o is not the last element of the sequence and $F(t_i^o) \neq 1$, we have the following property:*

$$G\left(t_{i+1}^o\right) + \beta t_i^o = G'\left(t_i^o\right) \cdot \frac{1 - F(t_{i-1}^o)}{f(t_i^o)} + \beta \frac{1 - F(t_i^o)}{f(t_i^o)} \tag{10.25}$$

where $G\left(x\right)$ is a convex cost function.

The proof follows the same principle as the one for Theorem 11.

Proposition 6. *For a smooth distribution with unbounded support and a convex cost function $G\left(x\right)$, solving* STOCHASTIC-CKPT *reduces to finding t_1^o that minimizes*

$$\sum_{i=0}^{\infty} \Big(G\left(t_{i+1}\right) + \beta t_i\Big) \mathbb{P}\left(X \geq t_i\right)$$

where $t_0^o = 0$, and for all $i \geq 2$,

$$t_i^o = G^{-1}\left(G'\left(t_{i-1}^o\right) \cdot \frac{1 - F(t_{i-2}^o)}{f(t_{i-1}^o)} + \beta\left(\frac{1 - F(t_{i-1}^o)}{f(t_{i-1}^o)} - t_{i-1}^o\right)\right) \tag{10.26}$$

For a smooth distribution with bounded support, the recurrence in Equation (10.26) still holds but the optimal sequence stops as soon as it reaches t_i^o with $F(t_i^o) = 1$.

Proof. Directly comes from rewriting Equation (10.25). □

10.3 Summary of contributions

We summarize in Table 10.1 our different contributions for both STOCHASTIC and STOCHASTIC-CKPT problems, with associated algorithms, theorems, and proofs in the manuscript. Finally, Table 10.2 recalls the different heuristics we introduced for STOCHASTIC-CKPT problem.

This concludes this chapter about algorithmic solution for STOCHASTIC-CKPT and STOCHASTIC. In the next chapter, we will propose a rigorous evaluation of all the proposed strategies of this chapter.

Problem	STOCHASTIC-CKPT			STOCHASTIC		
Distribution	Continous		Discrete	Continuous		Discrete
Support	Bounded	Unbounded	-	Bounded	Unbounded	-
Solution	$(1+\epsilon)$-approx	-	optimal	sub-optimal	sub-optimal	optimal
Algorithm Name	DYN-PROG-COUNT	-	DISCRETE-CKPT	BRUTE-FORCE	BRUTE-FORCE	NO-CHECKPOINT
Complexity	$\mathcal{O}\left(\frac{1}{\epsilon^3}\right)$	-	$\mathcal{O}(n^3)$	$\mathcal{O}(M)$	$\mathcal{O}(M)$	$\mathcal{O}(n^2)$
Ref. Proof	Theorem 5	-	Theorem 6	Proposition 4	Proposition 4	Theorem 13
Ref. Algorithm	Algorithm 4	-	Theorem 6	Algorithm 7	Algorithm 7	Theorem 13

Table 10.1: Summary of contributions to both STOCHASTIC and STOCHASTIC-CKPT problems, with the referred theorem and proof.

Name	ALL-CHECKPOINT	ALL-CHECKPOINT-PERIODIC	NO-CHECKPOINT-PERIODIC
Reference	Theorem 7	Algorithm 5	Algorithm 6
Distribution	Discrete	Discrete	Discrete
Problem	STOCHASTIC-CKPT	STOCHASTIC-CKPT	STOCHASTIC
Solution	Heuristic	Heuristic	Heuristic
Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 10.2: Summary of all heuristics under study for STOCHASTIC-CKPT problem.

Chapter 11

Performance Evaluation

Contents

In this chapter, we evaluate the algorithmic solutions for STOCHASTIC and STOCHASTIC-CKPT problems presented in Chapter 10.

The evaluations are processed in three steps:

- We first evaluate the solution without checkpointing of Section 10.2, and compare it with standard scheduling approaches. We perform this evaluation under two different cost models that cover both HPC and cloud computing systems. The evaluations are done by a simulation process, on synthetic applications (Section 11.1).
- In a second step, we provide through simulation an evaluation of the different strategies with checkpointing presented in Section 10.1. We also compare the performance to the non-checkpointing version and show the benefits of checkpointing (Section 11.2).

- Finally, we propose an evaluation of the solutions including checkpointing by performing real experiments with neuroscience application (Section 11.3).

We describe in Appendix A.2 the details of the software contributions for the different evaluations performed in this chapter.

11.1 A first evaluation: the model without checkpoint

In this section, we evaluate the different heuristics presented in Section 10.2.3 to solve STOCHASTIC problem. Our purpose in this section is to show that our contribution to solve STOCHASTIC, the BRUTE-FORCE algorithm, is a good solution. To do so, we evaluate BRUTE-FORCE in comparison with heuristics that do not use checkpointing, introduced in Section 11.1.1. In Section 11.2, we will compare solutions with and without checkpointing and show the benefits of checkpointing. It is expected that a good checkpointing solution should outperform the ones without checkpointing. This is why we perform a fair evaluation for BRUTE-FORCE as a first step in this dedicated section.

The code and setup of the experiments presented in this section are publicly available on `https://gitlab.inria.fr/vhonore/ipdps_2019_stochastic-scheduling`. Appendix A.2.1 presents some more details about this software production. In the following, the notion of performance stands for the expected cost of each algorithm under various job execution time distributions and cost functions.

11.1.1 Methodology

In this section, we present some simple heuristics that are inspired by common resource allocation strategies in the literature. These heuristics do not explore the structure of an optimal solution nor the probability distribution, but rely on simple incremental methods to generate reservation sequences.

In the following, we use $\mu = \mathbb{E}(X)$ to denote the mean of a given distribution, $\sigma^2 = \mathbb{E}(X^2) - \mu^2$ to denote its variance, and $m = Q(\frac{1}{2})$ to denote its median, where $Q(x) = \inf\{t|F(t) \geq x\}$ represents the quantile function. The different heuristics are defined as follows:

- MEAN-BY-MEAN: start with the mean (i.e., $t_1 = \mu$) and then make each subsequent reservation request by computing the conditional expectation of the distribution in the remaining interval, i.e.,

$$t_i = \mathbb{E}(X|X > t_{i-1}) = \frac{\int_{t_{i-1}}^{\infty} tf(t)dt}{1 - F(t_{i-1})}, \ \forall i \geq 2$$

  Deriving the sequence is straightforward for some distributions (e.g., exponential, uniform), but more involved for others. Full derivations for every considered distributions are described in Appendix B.

- MEAN-STDEV: start with the mean (i.e., $t_1 = \mu$) and then increment the reservation length by one standard deviation (σ) for each subsequent request, i.e.,

$$t_i = \mu + (i-1)\sigma, \ \forall i \geq 2$$

- MEAN-DOUBLING: start with the mean (i.e., $t_1 = \mu$) and then double the reservation length for each subsequent request, i.e.,

$$t_i = 2^{i-1}\mu, \ \forall i \geq 2$$

- MEDIAN-BY-MEDIAN: start with the median (i.e., $t_1 = m$) and then make each subsequent reservation request by using the median of the distribution in the remaining interval, i.e.,

$$t_i = Q(1 - \frac{1}{2^i}), \ \forall i \geq 2$$

The expected cost of each heuristic is obtained by using Equation 10.15. To get uniform results, we normalize the expected cost of each heuristic by the expected cost of an *omniscient* scheduler, which knows the job execution time t a priori, and thus would make a single request of length $t_1 = t$. Averaging over all possible values of t from the distribution $\mathcal{D}$, the omniscient scheduler that we denote by OMNISCIENT has an expected cost:

$$\mathbb{E}^o = \int_0^\infty (\alpha t + \beta t + \gamma) f(t)dt = (\alpha + \beta) \cdot \mathbb{E}[X] + \gamma$$

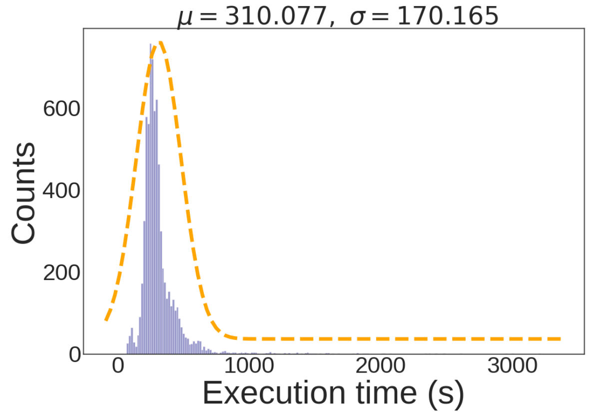
For a given distribution, the normalized ratio between algorithm performance and OMNISCIENT will always be larger than or equal to 1, and a smaller ratio means a better result.

We perform the evaluation of the heuristics under two different reservation-based scenarios, each associated to a different cost function:

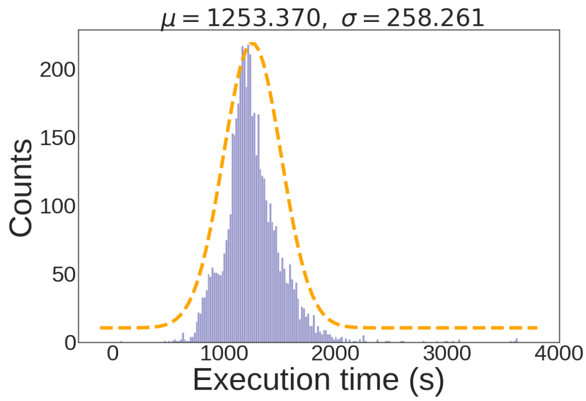
- RESERVATIONONLY (Section 11.1.2): This scenario is based on the *Reserved Instance* pricing scheme available in AWS [16], where the user pays exactly what is requested. Hence, we instantiate the cost function with $\alpha = 1, \beta = \gamma = 0$. For instance, such costs are incurred when making reservations of resources to schedule jobs on some cloud platforms, with hourly or daily rates. We consider nine probability distributions in this case, six of which have infinite support[1] and the remaining three have finite support. Table 11.1 lists these distributions with instantiations of their parameters used in the evaluation.

- HPC (Section 11.1.3): This scenario is based on executing large jobs on HPC platforms, where the cost, as represented by the total turnaround time of a job, is the sum of its waiting time in the queue and its actual execution time. We set $\beta = 1$ for

[1] We consider here one-side truncation for Truncated Normal distribution.



(a) Functional MRI quality assurance (fMRIQA) [58]



(b) Voxel-based morphometry quality assurance (VBMQA) [108]

Figure 11.1: Traces of over 5000 runs (histograms in purple) from July 2013 to October 2016 of two Neuroscience applications from the Vanderbilt's medical imaging database [71]. We fit the data to LogNormal distributions (dotted lines in orange) with means and standard deviations shown on top.

the execution time and instantiate the waiting time function (α, γ) by curve-fitting the data from the Intrepid machine, presented in Figure 9.1 of Chapter 9. The probability distribution of application walltime is derived from execution traces of Neuroscience applications as shown in Figure 11.1.

11.1.2 Results for ReservationOnly scenario

We first evaluate in Table 11.2 the performance of the two discretization schemes presented in Section 10.2.4 with different numbers of discrete samples. We can see that, for all distributions considered, the normalized costs with omniscient scheduler of both heuristics improve as we increase the number n of samples. The performance converges and gets close to that of Brute-Force when $n = 1000$, despite the differences in the convergence rate under different distributions and discretization schemes (Equal-time and Equal-Probability). Again, both heuristics take just a few seconds to run on an Intel i7 core, and the results provide good approximate solutions to the problem with sufficient samples. Even though performance of both schemes are close, we see that Equal-time seems to perform slightly better. Hence, we use Equal-time as discretization scheme in the remaining of this chapter, when using No-Checkpoint and heuristics presented in Table 10.2 of Chapter 10.

[2] This one-side truncation is used for evaluation of non-checkpointing strategies (Section 11.1)
[3] This two-side truncation is used for the evaluation of strategies with checkpointing (Section 11.2)

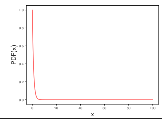
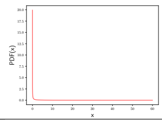
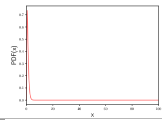
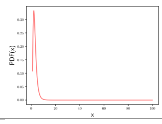
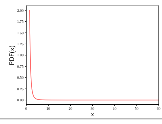
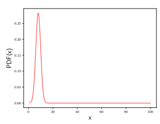
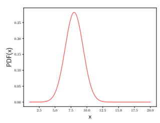
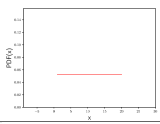
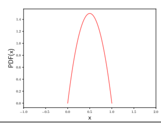
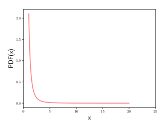
Distribution	**PDF $f(t)$**	**Instantiation**	**Support (in hours)**	PDF shape
Distributions with infinite support				
Exponential (λ)	$\lambda e^{-\lambda t}$	$\lambda = 1.0h^{-1}$	$t \in [0, \infty)$	
Weibull(λ, κ)	$\frac{\kappa}{\lambda}\left(\frac{t}{\lambda}\right)^{\kappa-1} e^{-\left(\frac{t}{\lambda}\right)^{\kappa}}$	$\lambda = 1.0h$ $\kappa = 0.5$	$t \in [0, \infty)$	
Gamma(α, β)	$\frac{\beta^\alpha}{\Gamma(\alpha)} t^{\alpha-1} e^{-\beta t}$	$\alpha = 2.0$ $\beta = 2.0h^{-1}$	$t \in [0, \infty)$	
Lognormal (ν, κ)	$\frac{1}{t\kappa\sqrt{2\pi}} e^{-\frac{(\ln t - \nu)^2}{2\kappa^2}}$	$\nu = 3.0h$ $\kappa = 0.5$	$t \in (0, \infty)$	
Pareto(ν, α)	$\frac{\alpha \nu^\alpha}{t^{\alpha+1}}$	$\nu = 1.5h$ $\alpha = 3.0$	$t \in [\nu, \infty)$	
TruncatedNormal(μ, σ^2, a)[2]	$\frac{1}{\sigma}\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2}}{1-\mathrm{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)}$	$\mu = 8.0$ $\sigma^2 = 2.0$ $a = 0.0$	$t \in [a, \infty)$	
Distributions with finite support				
Truncated Normal(ν, κ^2, a, b)[3]	$\frac{1}{\kappa}\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{t-\nu}{\kappa}\right)^2}}{1-\mathrm{erf}\left(\frac{a-\nu}{\kappa\sqrt{2}}\right)}$	$\nu = 8.0h$ $\kappa^2 = 2.0h^2$ $a = 1.0h$ $b = 20.0h$	$t \in [a, b]$	
Uniform(a, b)	$\frac{1}{b-a}$	$a = 1.0h$ $b = 20.0h$	$t \in [a, b]$	
Beta(α, β)	$\frac{t^{\alpha-1} \cdot (1-t)^{\beta-1}}{\mathrm{B}(\alpha, \beta)}$	$\alpha = 2.0$ $\beta = 2.0$	$t \in [0, 1]$	
Bounded Pareto(L, H, α)	$\frac{\alpha L^\alpha t^{-\alpha-1}}{1-\left(\frac{L}{H}\right)^\alpha}$	$L = 1.0h$ $H = 20.0h$ $\alpha = 2.1$	$t \in [L, H]$	

Table 11.1: Probability distributions and parameter instantiations.

Distribution	EQUAL-TIME							EQUAL-PROBABILITY						
	$n=10$	$n=25$	$n=50$	$n=100$	$n=250$	$n=500$	$n=1000$	$n=10$	$n=25$	$n=50$	$n=100$	$n=250$	$n=500$	$n=1000$
Exponential	2.61	2.40	2.33	2.33	2.39	2.35	2.31	3.68	2.76	2.56	2.59	2.28	2.34	2.36
Weibull	17.03	7.19	4.11	3.14	2.66	2.95	2.40	15.77	7.46	5.75	4.24	3.47	2.84	2.22
Gamma	2.22	2.17	2.17	2.13	2.12	2.08	2.20	2.66	2.39	2.38	2.23	2.27	2.27	2.13
Lognormal	1.93	1.86	1.96	1.89	1.93	1.96	1.87	2.93	2.52	2.18	2.00	1.92	1.91	1.93
TruncatedNormal	1.38	1.34	1.36	1.38	1.37	1.37	1.38	1.41	1.39	1.39	1.38	1.36	1.36	1.36
Pareto	31.54	13.02	6.88	3.80	2.09	1.74	1.71	32.05	12.99	3.76	5.09	2.97	1.99	1.66
Uniform	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33
Beta	1.82	1.82	1.81	1.86	1.78	1.79	1.79	1.79	1.82	1.78	1.81	1.79	1.81	1.80
BoundedPareto	2.18	1.88	1.84	2.04	1.98	1.91	2.00	2.59	2.17	1.90	1.99	1.91	1.94	1.91

Table 11.2: Normalized expected costs ($\tilde{\mathbb{E}}(S)/\mathbb{E}^o$) of the two discretization-based heuristics with different numbers of samples in the RESERVATIONONLY scenario.

Distribution	BRUTE-FORCE	MEAN-BY-MEAN	MEAN-STDEV	MEAN-DOUB.	MED-BY-MED	NO-CHECKPOINT
Exponential	2.15	2.36 (1.10)	2.39 (1.11)	2.42 (1.13)	2.83 (1.32)	2.31 (1.07)
Weibull	2.12	2.76 (1.30)	3.58 (1.69)	3.03 (1.43)	3.05 (1.44)	2.40 (1.13)
Gamma	2.02	2.26 (1.12)	2.18 (1.08)	2.24 (1.11)	2.51 (1.24)	2.20 (1.09)
Lognormal	1.85	2.19 (1.19)	2.09 (1.13)	1.95 (1.06)	2.30 (1.24)	1.87 (1.01)
TruncatedNormal	1.36	1.98 (1.46)	1.83 (1.35)	1.98 (1.46)	2.16 (1.60)	1.38 (1.02)
Pareto	1.62	1.82 (1.12)	2.18 (1.34)	1.75 (1.08)	2.26 (1.39)	1.71 (1.05)
Uniform	1.33	2.21 (1.66)	1.90 (1.43)	1.67 (1.26)	2.21 (1.66)	1.33 (1.00)
Beta	1.75	2.02 (1.15)	2.11 (1.20)	1.98 (1.13)	2.45 (1.40)	1.79 (1.02)
BoundedPareto	1.80	1.84 (1.02)	2.09 (1.16)	1.83 (1.01)	2.81 (1.56)	2.00 (1.11)

Table 11.3: Normalized expected costs of different heuristics in the RESERVATIONONLY scenario under different distributions. The values in the parenthesis show the expected costs normalized by those of the BRUTE-FORCE heuristic.

Table 11.3 presents, for each heuristic, the normalized expected cost, i.e., $\tilde{\mathbb{E}}(S)/\mathbb{E}^o$, under different probability distributions. The BRUTE-FORCE heuristic tries $M = 5000$ values of t_1, and EQUAL-TIME discretization procedure sets the truncation parameter to be $\epsilon = 10^{-7}$.

First, the normalized costs allow us to compare the performance of these heuristics with that of the omniscient scheduler to access the relative benefits of using *Reserved Instance* (RI) rather than *On-Demand* (OD). Indeed, if the per-hour price for RI is c_{RI} and the corresponding price for OD is c_{OD}, it is beneficial to use RI and compute a reservation sequence S, if $c_{\text{RI}} \cdot \tilde{\mathbb{E}}(S) \leq c_{\text{OD}} \cdot \mathbb{E}^o$, that is $\tilde{\mathbb{E}}(S)/\mathbb{E}^o \leq c_{\text{OD}}/c_{\text{RI}}$. In the case of Amazon AWS [16], the price for the two types of services can differ by a factor of 4, i.e., $c_{\text{OD}}/c_{\text{RI}} = 4$.

We can see in the table that the normalized costs of all heuristics satisfy $\tilde{\mathbb{E}}(S)/\mathbb{E}^o < 4$ for all distributions except for Pareto, which has a long tail[4] and thus incurs a higher cost for non-omniscient schedulers. Overall, the results show the benefit of using the reservation-based approach for the considered problem. We also observe that, compared with other heuristics, BRUTE-FORCE has better performance (see values in the parenthesis in the table), and this is because it computes a reservation sequence by exploring the properties of the optimal solution (Section 10.2.2).

We now study the BRUTE-FORCE heuristic in more detail. Table 11.4 shows the best t_1 found, which we denote by t_1^{bf}, and some other values of t_1 at different quantiles of the distributions with their normalized costs (in parenthesis). First, we can see that some values of t_1 can lead to invalid sequences that are not increasing (i.e., $t_{i+1} < t_i$ for some i), which are indicated by null cost values in the table. Moreover, even if the sequence is valid, compared to using t_1^{bf}, randomly guessing a t_1 can result in a cost that is not good enough in most cases. Although we can sometimes extract t_1's that could give a reasonable cost (e.g., in the case of Exponential distribution, $t_1 = 0$ results in a cost that is close to that given by t_1^{bf}), it is difficult in general to guess a good value for t_1 without using a systematic approach. We point out that more efficient algorithms may exist to search for the best t_1, but our BRUTE-FORCE procedure takes just a few seconds to run on an Intel i7 core with $M = 5000$, thus providing a practical solution that is close to the optimal for the problem.

11.1.3 Results for HPC scenario

We now present the evaluation results for the HPC scenario when using a real job execution time distribution under an HPC cost model. The distribution that we use (shown in Figure 11.1b) is generated from the execution traces of a Neuroscience application (VBMQA [28]). It follows a LogNormal law with parameters ($\mu = 7.1128h, \sigma = 0.2039h$) obtained by fitting the execution time data to the distribution curve, and this gives a mean of $\mu^d = 1253.37\text{s} \approx 0.348$ hour and a standard deviation of $\sigma^d = 258.261\text{s} \approx 0.072$ hour.

[4]Intuitively, a long-tail distribution has a large number of occurrences that are far from the beginning and central part of its support. Formally, it means that $\frac{1-F(x+y)}{1-F(x)} \to 1$ when $x \to \infty$, $\forall y > 0$.

Distribution	t_1^{bf} (assoc. cost)	Other values of t_1 (associated cost)			
		$Q(0.25)$	$Q(0.5)$	$Q(0.75)$	$Q(0.99)$
Exponential	0.73 (2.15)	0.29 (−)	0.69 (−)	1.39 (2.67)	4.61 (4.83)
Weibull	0.18 (2.12)	0.08 (2.51)	0.48 (2.35)	1.92 (3.87)	21.21 (13.49)
Gamma	1.23 (2.02)	0.48 (−)	0.84 (−)	1.35 (2.11)	3.32 (3.36)
Lognormal	29.64 (1.85)	14.34 (−)	20.09 (−)	28.14 (−)	64.28 (2.97)
TruncatedNormal	10.22 (1.36)	7.05 (−)	8.00 (−)	8.95 (−)	11.29 (1.42)
Pareto	2.59 (1.62)	1.65 (−)	1.89 (−)	2.38 (−)	6.96 (4.23)
Uniform	19.95 (1.33)	12.50 (−)	15.00 (−)	17.50 (−)	19.90 (−)
Beta	0.81 (1.75)	0.33 (−)	0.50 (−)	0.67 (−)	0.94 (1.89)
BoundedPareto	2.10 (1.80)	1.15 (−)	1.39 (−)	1.93 (−)	8.27 (4.64)

Table 11.4: The best t_1^{bf} found by the BRUTE-FORCE heuristic and other values of t_1 at different quantiles of the distributions with their normalized expected costs (in parenthesis) in the RESERVATIONONLY scenario.

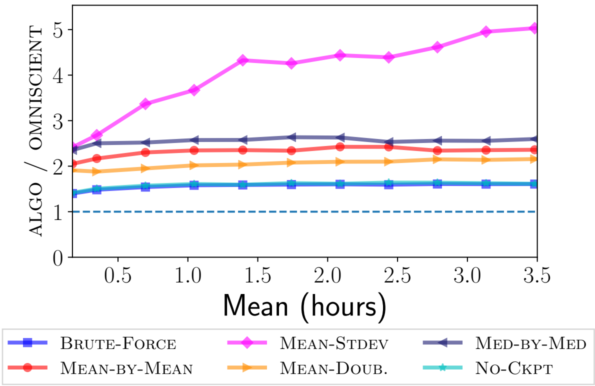
The average waiting time function (shown in Figure 9.1b) is obtained by analyzing the logs from 20 groups of jobs run on 409 processors of Intrepid [136] with different reservation requests. We get an affine function with parameters ($\alpha = 0.95, \gamma = 3771.84\text{s} \approx 1.05$ hour) obtained also by curve fitting. The execution time parameter is set to $\beta = 1$.

Figure 11.2 plots the normalized expected costs of different heuristics in this scenario. To evaluate the robustness of the results, we also vary the distribution parameters so that its mean and standard deviation are increased by up to a factor of 10 from their original values[5], i.e., up to $\mu^d \approx 3.48$ hours and $\sigma^d \approx 0.72$ hour. We can see from the figures that, regardless of the parameter variations, BRUTE-FORCE and the two discretization-based heuristics (EQUAL-TIME and EQUAL-PROBABILITY) have very close performance, which is significantly better than the performance of the other heuristics. The results are consistent with those observed in Section 11.1.2 for the RESERVATIONONLY scenario, and altogether they demonstrate the effectiveness and robustness of the proposed BRUTE-FORCE and discretization schemes for the STOCHASTIC problem.

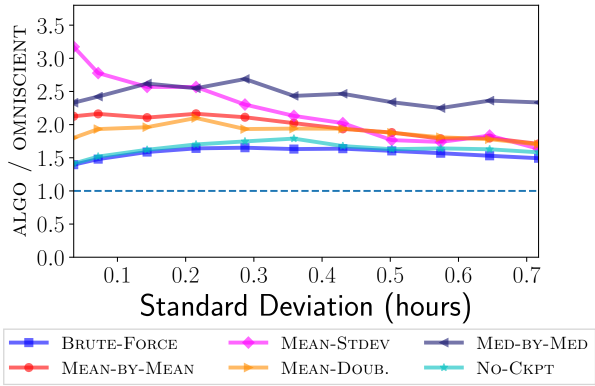
This concludes the evaluation of our proposed solutions for STOCHASTIC problem. Through an intensive set of simulation results, we demonstrated the efficiency of our strategies in comparison with state-of-the-art approach.

In the next section, we are interested in evaluating the strategies that includes checkpointing (STOCHASTIC-CKPT problem). It is expected that including checkpointing of

[5]Given a desired mean μ^d and a standard deviation σ^d, the LogNormal distribution can be instantiated with parameters $\sigma = \sqrt{\ln((\frac{\sigma^d}{\mu^d})^2 + 1)}$ and $\mu = \ln\left(\mu^d - \frac{{\sigma^d}^2}{2}\right)$.



(a)



(b)

Figure 11.2: Normalized expected costs of the different heuristics in the HPC scenario with different values for the mean (in hours) and standard deviation (in hours) of the LogNormal distribution ($\mu = 7.1128, \sigma = 0.2039$) with $\alpha = 0.95,\ \beta = 1.0,\ \gamma = 1.05$.

some well-chosen distribution should improve the performance of these strategies.

11.2 On the strategies with checkpointing

In this section, we evaluate the performance of the different heuristics of Section 10.1 in simulation. As we previously stated, performance stands for the expected cost of each algorithm under various job execution time distributions, C/R overhead and cost functions. We use jobs that follow a wide range of usual probability distributions as well as a distribution obtained from traces of a real Neuroscience application. We also show that checkpointing can be an efficient tool to optimize the performance of applications on reservation-based platforms, and under which conditions this statement holds. The code for this section is publicly available on `https://gitlab.inria.fr/vhonore/ckpt-for-stochastic-scheduling`. More details about the software development and the reproducibility of results are available in Appendix A.2.2.

11.2.1 Evaluation methodology

In this section, we evaluate five different algorithms from the following two sets of strategies:

- DYN-PROG-COUNT: This set includes Algorithm 4, and its ALL-CHECKPOINT and NO-CHECKPOINT variants described in Section 10.1.4.

- ALL-CHECKPOINT-PERIODIC: This set includes Algorithm 5, and its NO-CHECKPOINT-PERIODIC counterpart where checkpointing is not allowed (i.e., $\delta_i = 0, \forall i$).

The above algorithms are evaluated using two scenarios and one cost function called RESERVATIONONLY-PRICING. It follows the same principle as the one introduced in Section 11.1 for RESERVATIONONLY scenario. This function is based on the *Reserved Instances* pricing scheme in AWS [16], where the user pays exactly what is requested. Hence, $\alpha = 1, \beta = \gamma = 0$. For completeness, we present in Appendix C similar results with a second cost function called HPC-PRICING. It considers an additional cost that is proportional to the actual execution time (pay for what you use). Thus, $\alpha = 1, \beta = 1, \gamma = 0$.

We now detail the two scenarios we use to evaluate the algorithms, using RESERVATIONONLY-PRICING cost function:

- *Scenario 1* (Section 11.2.2): We consider nine usual probability distributions, five of which have infinite support (Exponential, Weibull, Gamma, LogNormal, Pareto) and four have finite support (Truncated Normal, Uniform, Beta, Bounded Pareto). We use the same instantiation of the distributions as the ones presented in Table 11.1 The first five distributions are truncated and fed as input to Algorithm 4. To do so, we set the upper bound of the infinite support to $b = Q(1 - \upsilon)$, where $Q(x) = \inf\{t|F(t) \geq x\}$ is the quantile function and υ is a small constant. In our simulation, we set $\upsilon = 10^{-7}$. During the discretization procedure in Algorithm 4, we then normalize the probabilities of all discrete values so that they sum to 1. We set $C = R = 360$ seconds (0.1 hour). This checkpointing cost is extracted from [81] and corresponds to an average checkpointing duration, where an optimistic one is 60 seconds and a pessimistic one is 600 seconds. We further discuss the impact of the checkpointing cost on the performance.

- *Scenario 2* (Section 11.2.3): In this scenario, we consider the execution time traces of a real Neuroscience application, and fit a LogNormal distribution to its execution times. To further evaluate the robustness of the algorithms, we perturb the parameters of the fitted distribution by varying its mean and standard deviation and show the impact on the performance.

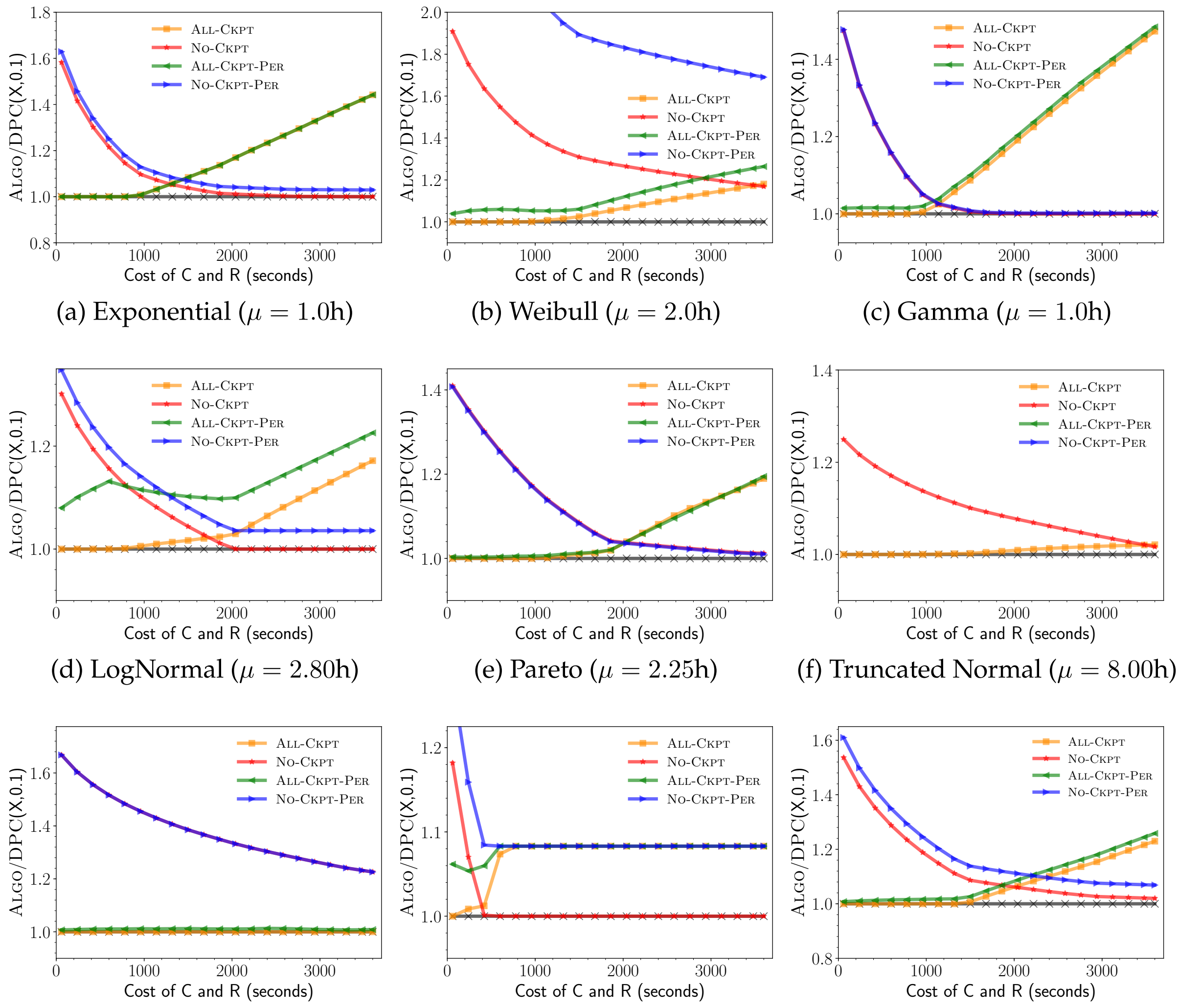
11.2.2 Results for Scenario 1

We first evaluate the performance of DYN-PROG-COUNT compared to the other strategies, when the values of R and C varies. Figure 11.3 presents the performance of these strategies normalized to that of DYN-PROG-COUNT (black line for $y = 1.0$) for all distributions of Table 11.1 using RESERVATIONONLY-PRICING cost function. We use $\varepsilon = 0.1$ for DYN-PROG-COUNT and its variants. Regarding periodic strategies, we choose the best value for the number of chunks τ in $[1, 1000]$. Not surprisingly, we can observe that when C and R are small, the best result is to use the ALL-CHECKPOINT strategy while when they are large, one should use the NO-CHECKPOINT strategy. There exist thresholds on the sizes of C and R where DYN-PROG-COUNT uses a mix of checkpointed and not checkpointed reservations. In that case, the gain of using DYN-PROG-COUNT can be up to 10% in compared with its variants. An interesting future direction is to find properties on those threshold depending on the distribution. Finally, one should observe that the gain obtained with DYN-PROG-COUNT compared to the best periodic solution is in general more important (for Truncated Normal, the performance of periodic solutions are worse than a factor 2 of DYN-PROG-COUNT). For Exponential distribution, ALL-CHECKPOINT and its periodic counterpart are identical (proof can be found in Section 10.1.4), due to the memoryless property of the exponential distribution.

We then study the impact of ε on the performance of DYN-PROG-COUNT (DPC) when $R = C = 6$min, 30min and 60min. The idea is that when $\varepsilon = 1$, this theoretically guarantees that the performance is at most twice ($= 1+\varepsilon$) that of the optimal, but in practice it can be a lot better. We study in Figure 11.4 the performance of DYN-PROG-COUNT for various values of ε for distributions of Table 11.1 with RESERVATIONONLY-PRICING cost function. All performance are normalized by DYN-PROG-COUNT for $\varepsilon = 0.1$. We can see that in practice, the convergence to the lower bound in performance is fast. Indeed, for $\varepsilon = 1$ and $C = R = 6$min (Figure 11.4a), almost all distributions already reach convergence, except for Weibull and Pareto (which have a much larger domain of definition and specific properties[6]). For those distributions, we see that they converged for $\varepsilon = 0.1$. We observe similar trends in Figures 11.4b and 11.4c when R and C increases. Interestingly, one can note that Pareto distribution converges faster for $C = R = 30$ or 60min than for $C = R = 6$min, while Weibull distribution shows contrary behavior. Convergence is still achieved for $\varepsilon = 0.1$. This shows the possible impact of application features on algorithm behavior.

Our final evaluation for this scenario is a study of the impact of the size of the period. Until now we have always chosen the period that minimized the objective functions. Table 11.5 shows the performance of both variants of the periodic algorithms, ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized by that of DYN-PROG-COUNT ($\varepsilon = 0.1$), when $C = R = 360$s using RESERVATIONONLY-PRICING cost function. For each distribution: the second columns shows the best period found when τ varies from 1 to 1000 (with its associated cost normalized by that of DYN-PROG-COUNT), and the other columns present results for specific values of τ in that interval. As was observed before, ALL-CHECKPOINT-PERIODIC is in general not able to match DYN-PROG-COUNT (except for some distributions). We can also clearly see that NO-CHECKPOINT-PERIODIC performs even worse than ALL-CHECKPOINT-PERIODIC. The reason is that the checkpointing cost is relatively low in this setup, so it is preferable to checkpoint more often than never. Hence, when

[6] For instance, Pareto is a long-tail distribution, meaning that it has a large number of occurrences that are far from the beginning and central part of its support. Formally, it means that $\frac{1-F(x+y)}{1-F(x)} \to 1$ when $x \to \infty$, $\forall y > 0$.

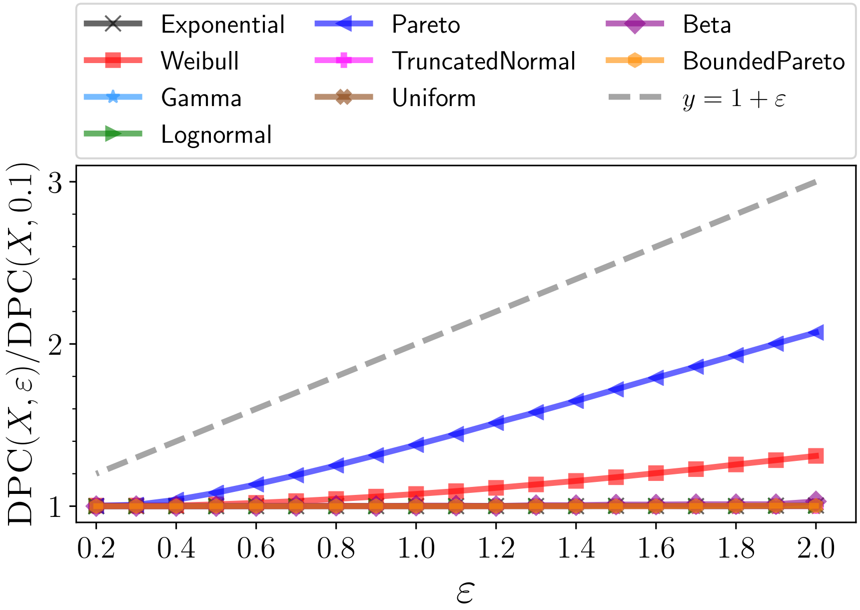


(a) Exponential ($\mu = 1.0$h) (b) Weibull ($\mu = 2.0$h) (c) Gamma ($\mu = 1.0$h)

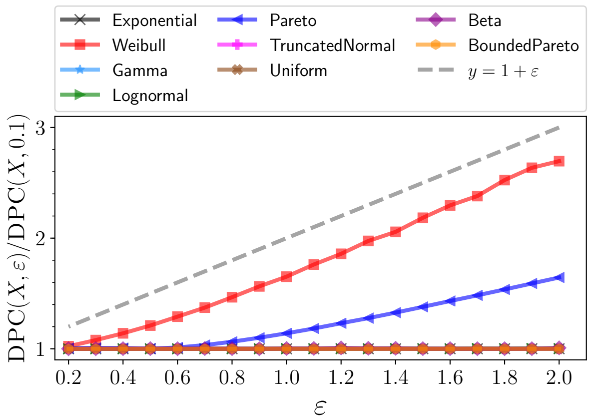
(d) LogNormal ($\mu = 2.80$h) (e) Pareto ($\mu = 2.25$h) (f) Truncated Normal ($\mu = 8.00$h)

(g) Uniform ($\mu = 10.5$h) (h) Beta ($\mu = 0.5$h) (i) Bounded Pareto ($\mu = 1.84$h)

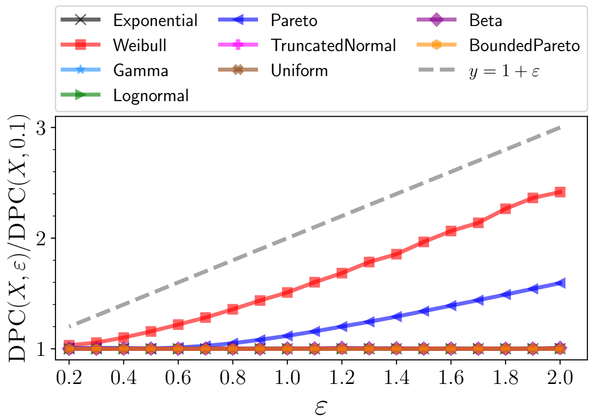
Figure 11.3: Expected costs of the different strategies normalized to that of DYN-PROG-COUNT$(X, 0.1)$ when $C = R$ vary from 60 to 3600 seconds, for all distributions in Table 11.1 with support considered in hours with RESERVATIONONLY-PRICING cost function. We indicate in brackets the mean μ of each distribution.



(a) $C = R = 6\text{min}$



(b) $C = R = 30\text{min}$



(c) $C = R = 60\text{min}$

Figure 11.4: Expected cost of DYN-PROG-COUNT(X, ε) as a function of ε for different distributions for X with RESERVATIONONLY-PRICING cost function. $C = R$ are set to $6, 30$ and 60min.

$C = R = 1$h, Table 11.6 shows that NO-CHECKPOINT-PERIODIC performs slightly better than ALL-CHECKPOINT-PERIODIC. Finally another observation is that a wrong period size can significantly deteriorate the performance of the periodic algorithms.

Distribution	ALL-CHECKPOINT-PERIODIC							NO-CHECKPOINT-PERIODIC						
	Best τ	$\tau = 1$	$\tau = 200$	$\tau = 400$	$\tau = 600$	$\tau = 800$	$\tau = 1000$	Best τ	$\tau = 1$	$\tau = 200$	$\tau = 400$	$\tau = 600$	$\tau = 800$	$\tau = 1000$
Exponential	27 (1.00)	9.82	2.31	4.02	5.75	7.47	9.19	14 (1.38)	9.82	6.91	13.23	19.56	25.89	32.22
Weibull	380 (1.06)	106.50	1.15	1.06	1.10	1.18	1.27	89 (2.54)	106.50	3.26	5.32	7.52	9.74	11.98
Gamma	14 (1.02)	6.02	3.68	6.76	9.85	12.93	16.02	8 (1.26)	6.02	9.34	18.08	26.82	35.56	44.31
Lognormal	11 (1.11)	3.60	3.92	7.05	10.18	13.32	16.45	4 (1.25)	3.60	15.48	30.17	44.86	59.56	74.25
Pareto	1000 (1.01)	228.77	1.75	1.23	1.09	1.03	1.01	562 (1.33)	228.77	1.78	1.37	1.33	1.37	1.45
TruncatedNormal	2 (2.15)	2.18	8.17	14.31	20.45	26.59	32.73	1 (2.18)	2.18	197.54	394.01	590.47	786.93	983.39
Uniform	8 (1.01)	1.57	3.17	5.51	7.86	10.20	12.54	1 (1.57)	1.57	51.08	101.33	151.59	201.84	252.10
Beta	2 (1.06)	1.11	30.78	61.00	91.23	121.45	151.67	1 (1.11)	1.11	40.85	81.15	121.45	161.75	202.05
BoundedPareto	32 (1.01)	7.53	1.73	2.71	3.70	4.70	5.69	14 (1.44)	7.53	6.51	12.28	18.06	23.83	29.61

Table 11.5: Expected cost of ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized by DYN-PROG-COUNT$(X, 0.1)$ for $C = R = 360$s, with RESERVATIONONLY-PRICING cost function.

Distribution	ALL-CHECKPOINT-PERIODIC							NO-CHECKPOINT-PERIODIC						
	Best τ	$\tau = 1$	$\tau = 200$	$\tau = 400$	$\tau = 600$	$\tau = 800$	$\tau = 1000$	Best τ	$\tau = 1$	$\tau = 200$	$\tau = 400$	$\tau = 600$	$\tau = 800$	$\tau = 1000$
Exponential	12 (1.44)	7.35	9.49	18.53	27.57	36.61	45.66	14 (1.03)	7.35	5.17	9.90	14.64	19.38	24.12
Weibull	156 (1.26)	70.94	1.29	1.65	2.11	2.60	3.11	89 (1.69)	70.94	2.17	3.54	5.01	6.49	7.98
Gamma	7 (1.48)	4.77	17.59	34.70	51.82	68.93	86.05	8 (1.00)	4.77	7.40	14.32	21.25	28.18	35.11
Lognormal	4 (1.23)	2.98	18.81	36.96	55.11	73.26	91.42	4 (1.04)	2.98	12.82	24.98	37.14	49.30	61.47
Pareto	563 (1.20)	175.44	1.58	1.24	1.21	1.24	1.31	562 (1.02)	175.44	1.37	1.05	1.02	1.05	1.11
TruncatedNormal	1 (1.85)	1.85	38.01	74.45	110.89	147.33	183.78	1 (1.85)	1.85	167.50	334.08	500.67	667.25	833.83
Uniform	3 (1.01)	1.23	13.46	26.26	39.07	51.88	64.69	1 (1.23)	1.23	39.89	79.13	118.37	157.61	196.85
Beta	1 (1.08)	1.08	207.32	414.09	620.87	827.64	1034.42	1 (1.08)	1.08	39.93	79.31	118.70	158.08	197.47
BoundedPareto	14 (1.26)	5.59	5.72	10.88	16.05	21.21	26.38	14 (1.07)	5.59	4.82	9.11	13.39	17.68	21.96

Table 11.6: Expected cost of ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized by DYN-PROG-COUNT$(X, 0.1)$ for $C = R = 1$h, with RESERVATIONONLY-PRICING cost function.

11.2.3 Results for Scenario 2

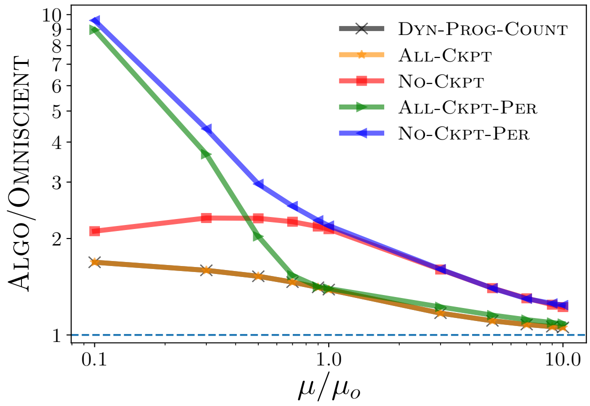
We now present the simulation results for a probability distribution fitted to the execution time traces of a real Neuroscience application (*a code for structural identification* *of orbital anatomy*) extracted from the Vanderbilt's medical imaging database [71]. Figure 9.2 shows the execution time traces of the application and its fitted LogNormal distribution. Figure 11.5 presents the performance of different algorithms for this fitted distribution using the RESERVATIONONLY-PRICING cost function, for different values of $C = R$. To evaluate the robustness of algorithms, we also vary the original mean μ_o (Figures 11.5a 11.5c 11.5e) or standard deviation σ_o (Figures 11.5b 11.5d 11.5f) of the distribution from their original values[7]. For readability, all axis are in logscale. We fix $\varepsilon = 1.0$ and test checkpoint/restart costs equals to 6min, 1h and 12h. For periodic strategies, we use similar brute-force procedure as Scenario 1 to find the period that performs best. The expected costs of the algorithms are normalized with *omniscient* scheduler (blue dashed line), as described in Section 11.1.1.

We can observe that DYN-PROG-COUNT always gives the best performance. As previously observed, the checkpointing cost influences the performance of NO-CHECKPOINT and ALL-CHECKPOINT with regard to DYN-PROG-COUNT. When $C = R = 600$ seconds or $C = R = 3600$s (Figure 11.5a - 11.5d), the value is low enough to allow for checkpointing all reservations, the performance of DYN-PROG-COUNT and ALL-CHECKPOINT are the same and outperforms NO-CHECKPOINT by a wide margin. However, when the checkpoint/restart overhead increases to 12h (roughly $\frac{\mu_o}{2}$), we see that checkpointing all reservations become over-costly (NO-CHECKPOINT is better than ALL-CHECKPOINT). In that case, DYN-PROG-COUNT outperforms all other strategies. One can observe that when the ratio μ/σ is large (either by increasing the mean, or decreasing the standard deviation), the solutions converge to the omniscient scheduler. This could be expected, in this case the variability becomes negligible and the job behaves similarly to a deterministic job. As for the periodic algorithms, ALL-CHECKPOINT-PERIODIC has better performance than NO-CHECKPOINT-PERIODIC. However, both algorithms have worse performance than DYN-PROG-COUNT. The results demonstrate the robustness of DYN-PROG-COUNT for a practical application with different distribution parameters. We see that, for this instantiation, it is always beneficial to use RI rather than OD when using the BRUTE-FORCE heuristic. Hence, our reservation-based approach is validated in terms of cost model in Cloud Computing.

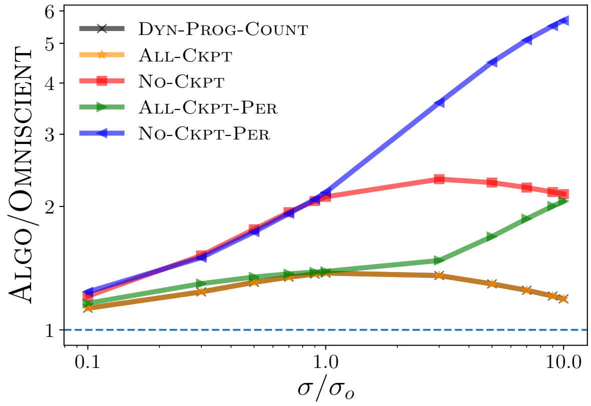
11.3 Experiments for Checkpointing Strategies

In this section, we conduct real experiments on an HPC platform by using three stochastic Neuroscience applications. The goal of the experiments is to study the performance of different reservation and checkpointing strategies when scheduling multiple jobs in a shared HPC execution environment.

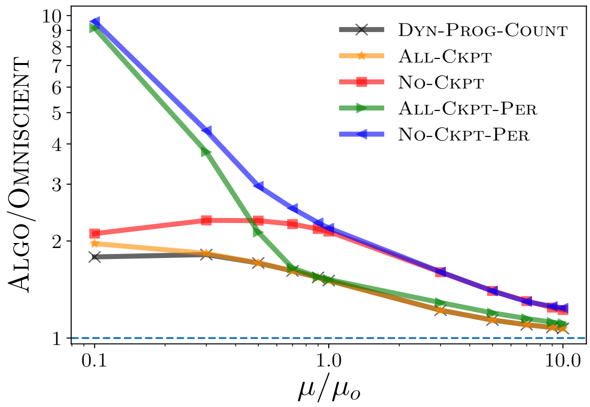
[7]Given a desired mean μ and a desire standard deviation σ, the LogNormal distribution can be instantiated with parameters $\kappa = \sqrt{\ln\left((\frac{\sigma}{\mu})^2 + 1\right)}$ and $\nu = \ln\left(\mu - \frac{\kappa^2}{2}\right)$.



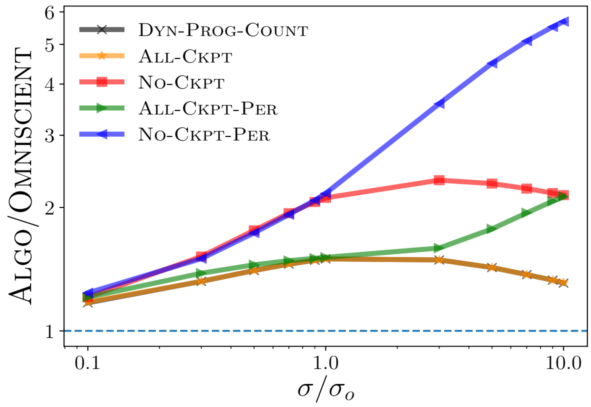
(a) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 600$s



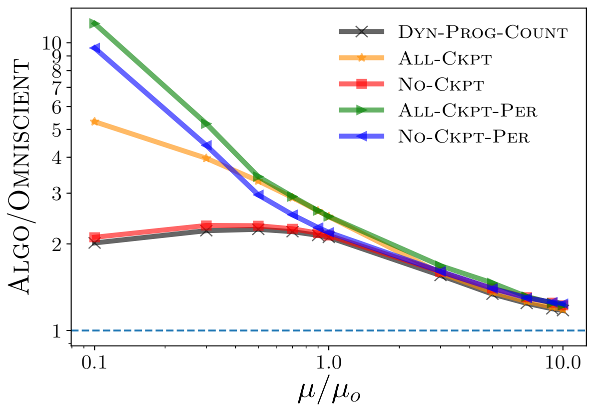
(b) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 600$s



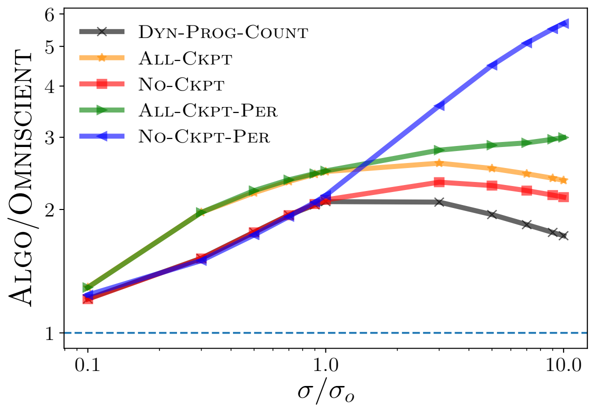
(c) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 1$h



(d) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 1$h



(e) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 12$h



(f) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 12$h

Figure 11.5: Normalized performance of algorithms with omniscient scheduler when μ or σ vary, using RESERVATIONONLY-PRICING cost function ($\alpha = 1.0$, $\beta = \gamma = 0$). Basis is the LogNormal distribution in Figure 9.2 ($\mu_o = 21.4$h, $\sigma_0 = 19.7$h). $C = R$ are set to $600, 3600$ and 43200s (12h), $\varepsilon = 1$.

11.3.1 Experimental setup

The chosen Neuroscience applications are described in Table 11.7 along with their execution characteristics, which are extracted from the Vanderbilt's medical imaging database [71]. In particular, the walltime distributions are obtained by fitting the execution time traces, and the checkpointing/restart costs are obtained by analyzing and averaging their memory footprints. Note that, for these applications, the restart costs (R) are different from the checkpointing costs (C).

Here, we focus on the evaluation of the following two different sets of strategies:

- An HPC-for-Neuroscience strategy (called HPC in Section 11.3.2), which uses the average of the last 5 runs as the initial reservation length and then increases it by a factor of 1.5 for each subsequent reservation. This strategy is currently used by the MASI group [91] at Vanderbilt to handle stochastic Neuroscience applications.

- Our proposed DYN-PROG-COUNT strategy and its ALL-CHECKPOINT variant.

We ran the experiments on a 256-thread Intel Processor (Xeon Phi 7230, 1.30GHz) while submitting jobs through the Slurm scheduler [155]. All three Neuroscience applications are sequential (i.e., uses a single hardware thread) and perform some medical imaging analysis. The variation in the execution time is due to the different characteristics of the input data. However, as we do not have access to the raw input images, we used the information in the logs to simulate the characteristics of the input data, thereby forcing a job to run for a certain walltime and saving a specific amount of data for the checkpoints. In each experiment, we submitted 500 jobs from one of the three applications, and recorded the completion time of each job. We use the average job *stretch* (defined as the ratio between the total execution time of a job and its actual walltime) to show the individual job performance, and use the *utilization* (defined as the ratio between the sum of all jobs' walltimes and the total time required to execute them) to show the performance of the system for the whole job set. During the experiments, the scheduler has complete access to the entire platform, thus corresponding to the scenario with $\alpha = 1$, $\beta = 0$, $\gamma = 0$.

11.3.2 Experimental results

By experimenting on a real system, we investigate the robustness of our strategy: 1) when multiple applications are running concurrently; 2) when the read/write times vary due to congestion while accessing I/O and/or due to application interference; 3) when the C/R costs vary depending on when in the application the checkpoint/restart takes place (i.e., different values for different reservations). Figure 11.6 shows the performance of the three strategies when submitting 500 jobs from each application to the Slurm scheduler. We manually force the C/R costs to be the same (as in Table 11.7) for

Application Type	**Walltime distribution**	C	R
Diffusion model fitting (Qball)	Gamma (k = 1.18, θ = 34, $[a, b] = [146s, 407s]$)	90s	40s
Diffusion model fitting (SD)	Weibull (k = 1043811, λ = 1174322466, $[a, b] = [46min, 2.3h]$)	25min	10min
Functional connectivity analysis (FCA)	Gamma (k = 3.6, θ = 72, $[a, b] = [165s, 1003s]$)	150s	100s

Table 11.7: Characteristics of the chosen Neuroscience applications.

each strategy so as to study the effects of application interference and the runtime system's performance variability on our model. The findings are consistent with the simulation results (in Section 11.2), showing that DYN-PROG-COUNT performs better than its ALL-CHECKPOINT variant in terms of both system utilization and average job stretch using all three applications. Moreover, the two algorithms outperform the simple HPC strategy.

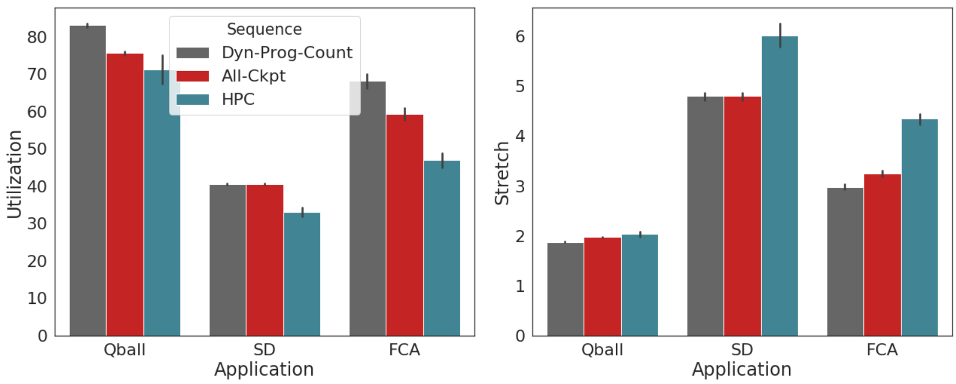


Figure 11.6: Utilization and average job stretch for DYN-PROG-COUNT, ALL-CHECKPOINT and the HPC strategies.

Depending on when the checkpoint is being taken, the checkpoint size and thus the time to save and restore the application can vary. Figure 11.7 shows the results when the C/R costs could vary for different reservations. Based on the log traces of these three applications, we noticed that their memory footprints can vary by as much as 30% depending on when the checkpoint is taken (e.g., the checkpoint time can vary between 80 and 110 seconds for Qball). Our experiment generates random checkpoint sizes using a uniform distribution with the mean given by the average checkpoint size from the traces, and forces the application to read/write the corresponding amount at the beginning/end of the execution. In this experiment, we assume that the checkpointing time is included in the request time and is never responsible for applications exceeding their allocated time. While the DYN-PROG-COUNT solution is computed using the average C/R costs presented in Table 11.7, the experimental results show that its performance is robust up to 15-20% variability in the C/R costs. Moreover, the average job stretch appears to be even more stable than the utilization, suggesting that most of the submitted jobs are not impacted by the fluctuation in the C/R costs.

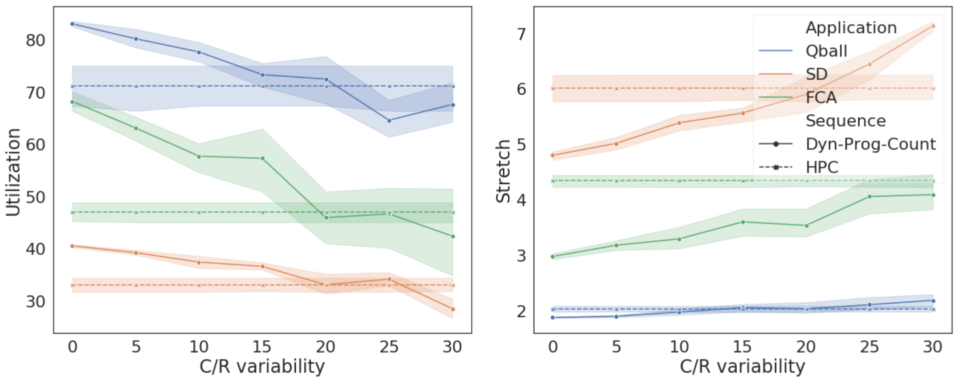


Figure 11.7: Utilization and average job stretch for the three applications (blue: Qball; Orange: SD; Green: FCA) when varying the C/R costs by different percentages (0 to 30%) using the DYN-PROG-COUNT strategy. Horizontal lines represent the results for the HPC strategy.

If application-level checkpoint is used, the application is usually aware of the checkpoint size, thus the checkpointing process can start before the reservation is over. The subsequent submissions can easily adapt to this deviation with the first checkpoints that are smaller than the one used to compute the sequence (this is the case for Figure 9). For system-level checkpoint, the application footprint usually remains similar throughout the execution of the application. In case the checkpointing time is causing the application to exceed the reserved time, the submission will fail and subsequent submissions can take this into account by adding the wasted time.

The limitation of our method is visible for applications with large variability in checkpointing size, which can be due to multiple factors, either within the application that presents different memory footprints throughout its execution, or by system-level causes, such as I/O congestion or failures. Such large variability in checkpointing size compared to what is used to compute the reservation sequence can result in worse performance when using our method, and the classic HPC model would be preferred in this case. This limitation is discussed in Chapter 12 where we investigate methods to incorporate variation of checkpointing size into the computation of the reservation sequence, by either using historic information or adapting the subsequent request times based on the sizes of previous checkpoints. Variation of C/R times is one of the major

DYN-PROG-COUNT		HPC		Improvement	
Utilization	**Avg Stretch**	**Utilization**	**Avg Stretch**	**Utilization**	**Avg Stretch**
67	2.04	55	2.34	21%	15%
73	1.72	62	2.04	18%	19%
62	2.08	55	2.46	12%	18%
71	1.88	64	2.1	11%	12%
63	2.19	56	2.41	11%	10%
71	1.74	64	1.96	10%	12%
75	1.51	68	1.69	10%	12%
68	2.09	65	2.19	4%	5%
61	2.24	60	2.32	2%	4%
77	1.96	75	1.99	2%	2%

Table 11.8: Utilization and average job stretch for 10 different runs, each using 500 jobs from all three applications. The runs are ordered by the best improvement of DYN-PROG-COUNT in utilization.

future research direction to improve the strategies with checkpointing.

Finally, we conduct experiments in a more realistic scenario by running all three applications at the same time and investigating the impact on the strategies. In particular, we submitted a total of 500 jobs (100 from Qball, and 200 each from SD and FCA), and kept the C/R costs constant across different reservations. We recorded the utilization and average job stretch when using DYN-PROG-COUNT compared to the HPC strategy for 10 different runs choosing different instances from the traces each time. The results are presented in Table 11.8. We can see that DYN-PROG-COUNT improves both utilization and average job stretch by 10% on average, and by up to 20% depending on the instances submitted. Overall, these results again illustrate the robustness of our algorithm and confirm its benefit for scheduling stochastic applications on reservation-based platforms, as long as checkpoint costs remain constant for each application.

Conclusion

In this section, we evaluated our scheduling strategies using both simulations and experiments. We demonstrated the efficiency of the different solutions compared to state-of-the-art heuristics, with different cost models covering a wide range of possible scenarios. We also exhibited the benefits of using checkpointing at the end of some well-chosen reservations in order to save the progress of the application through the successive reservations.

One of the main limitation of our checkpointing model is that it assumes a constant overhead for checkpointing and restarting. In practice, this may not be really representative of the application behavior. In Chapter 7, we mentioned that we could estimate the peak memory of the application, hence asking for the minimal amount of resources that satisfies this peak memory. However, if one could observe peaks in memory consumption, it means that at some moment of the application execution, the memory consumption may be much lower than the peak. Thus, checkpointing at such a moment should be more cost-efficient and then may impact the checkpointing decisions of our strategies.

In Chapter 12, we will propose a more in-depth study of the SLANT application, by observing the behavior at the task-level. From our new observations, we will refine our application model, and we will design sequences with memory-aware checkpointing decisions.

Chapter 12

One Step Further: Profiling Applications for Better Strategies

Contents

In previous chapters, we presented reservation-based strategies for stochastic applications on HPC or cloud infrastructures. To tackle the challenge of walltime variability, we provided solutions with either checkpointing to save progress of application at the end of some well-chosen reservations, or without in case checkpointing is not available for the target application. So far, we assumed that the execution time of these applications can be modeled by a (known) probability distribution. Also, we assumed a flat memory model for the application so that the cost of checkpoint-restart is constant wherever a checkpoint is performed during application execution.

The validity of this assumption is critical as our proposed solutions require the knowledge of the checkpoint (C) and restart (R) costs in order to determine the size of the successive reservations. If C is much more important than expected, the checkpoint process fails due to lack of time in the reservation. Hence, this failure directly impacts the upcoming reservations and so, the performance of the overall strategy. At the contrary, if R (and/or C) is lower than estimated, the reservation ends earlier and there is some reserved time that has been wasted.

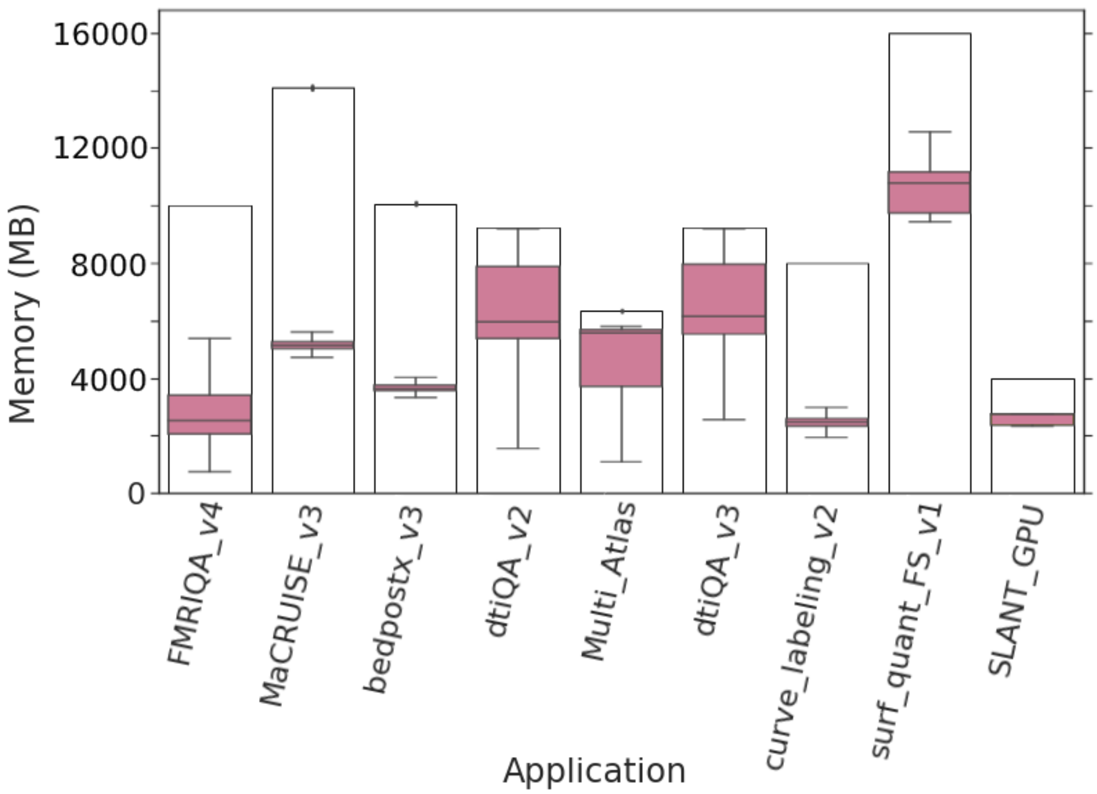


Figure 12.1: Memory requests during submission and memory usage variation for nine representative medical and neuroscience applications.

Figure 12.1 presents a first analysis of the memory requirements and requests for nine exploratory applications from the medical and neuroscience department at the Vanderbilt University [91] that we collaborated with to obtain historic of application runs. The logs are generated for a 6-month period in 2018 running on their in-house cluster. Users often utilize only fractions of the requested memory (e.g. MaCRUISE_v3, bedpostx_v2 in Figure 12.1) or end up with their application killed due to memory underestimation (e.g. dtiQA_v2 and dtiQA_v3). Users tend to overestimate their resource requirements in both time and memory, which leads to these application typically waiting in the scheduler queue for days before eventually running. In addition, the stochastic memory utilization often requires users to request only high memory nodes for their execution.

We see here a big contrast with applications studied in Part I. Memory-intensive applications we mentioned were not subject to run-to-run variability in memory needs, due to their monolithic and static code structure. We had applications with important memory needs, but constant over the execution. Hence, one can reserve some resources (matching minimum memory requirements) on the target machine for a known duration of time depending on the considered application. Then, resources are distributed between the different parallel tasks of the big block of code in order to optimize the total execution time of the applications.

Overall, based on the feedback from application users, it seems that stochastic applications present a stochastic profile for both execution time and memory features. In this chapter, we investigate the correctness of this observation. From a memory perspective, we want to verify the correctness of the flat memory model assumption that we expressed in previous chapters. To do so, we propose a more fine-grained study of an exploratory stochastic application, SLANT, that we already studied in Chapter 7. The aim of this chapter is to verify the applicability in practice of strategies presented in Chapter 9. Specifically, we are interested in verifying the assumption that checkpoint cost is constant through application execution. We show that this is not the case for SLANT application, due to the memory footprint of the application exhibiting many variations.

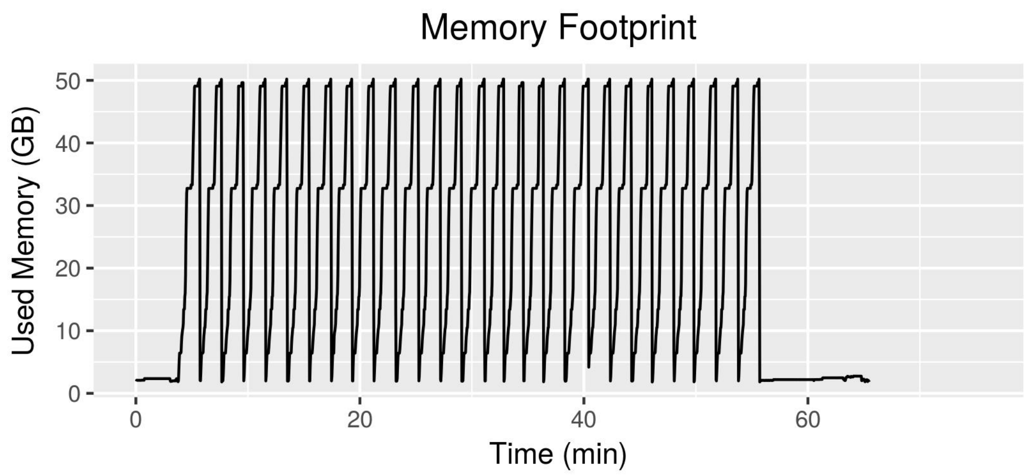
Based on our observations of SLANT, we then propose a generic application model where an application is described as a chain of tasks whose walltimes follow probability distributions. We use this model to estimate the resource request for SLANT when deployed on an HPC system. We also show that our resource estimator needs only a few runs to learn the model and to optimize the submission and execution of these types of applications without any modification to the batch scheduler or HPC middleware. This is essential for productivity-focused applications since their codes are in continuous change based on the requirements of each study. Performance prediction methods can be used by scientific applications to adjust their resource requirements during submission. However they tend to work well only on well known codes that can provide a rich history of past runs. Our study aims to bridge the gap between the specific characteristics of exploratory applications and the strict requirements of HPC batch schedulers that hinder productivity and innovation for new computational methods. We then propose memory-aware reservation strategies inspired from the ones presented in Section 10.1 of Chapter 10. Finally, we evaluate these new strategies and give perspectives to this work.

The contributions of this chapter are:

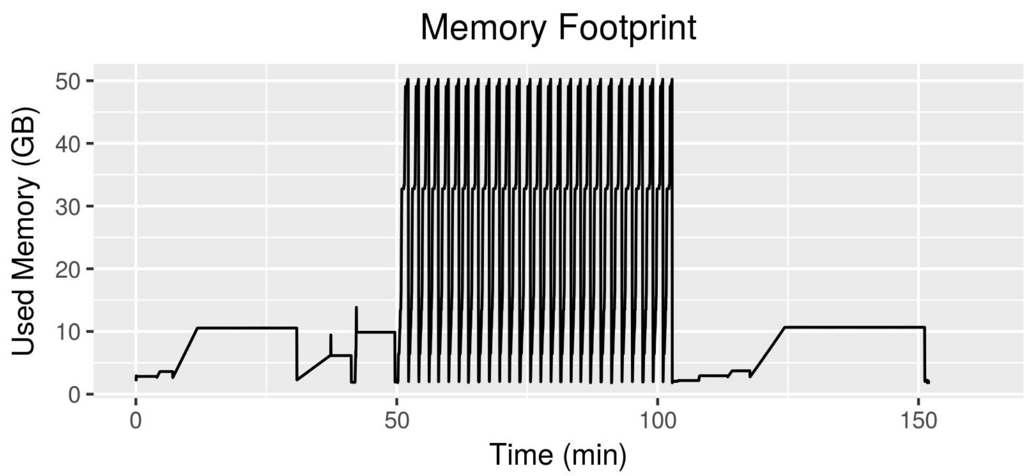
- A complete characterization of the SLANT application, from both task and memory perspectives
- A refined application model and memory-aware scheduling strategies for STOCHASTIC-CKPT
- An evaluation of these new strategies via different metrics

12.1 Task-level observations

In this section, we continue the study of the SLANT application initiated in Chapter 7. We previously mentioned that studies using machine learning methods to estimate the future resource consumption of an application assume a constant peak memory footprint (e.g. [134]). We assumed that this could be used as constant value for checkpointing overhead. In this section, we study more closely the memory behavior of these new HPC applications.



(a) Typical memory profile with OASIS input.



(b) Typical memory profile with DRD input.

Figure 12.2: Examples of memory footprints of the SLANT application with inputs from each considered dataset. Memory consumption is measured every 2 seconds with the *used memory* field of the `vmstat` command.

Figure 12.2 presents the memory footprint of two runs of the SLANT application, one for each of the input categories. Note that all other runs follow similar trends, specifically the peak memory usage is not dependent on the input, only the time depends (and hence the average memory utilization). For both profiles, we can see clearly the three phases of the application (pre-processing, deep-learning, post-processing). Note that these traces hint at the fact that the difference in executed time is more linked to a quality element since there is fewer pre/post-processing time for OASIS inputs.

In the following, we focus our discussions on the runs obtained from the 88 DRD inputs (Figure 12.2b) because their pre/post processing steps are more interesting, although the same study could be done for the OASIS inputs.

These memory footprints show that the runs can be divided into roughly seven different tasks of "constant" memory usage:

- *pre-processing* phase: This phase includes the four first tasks. The 1st task shows a memory consumption peak of around 3.5GB for the few first minutes of the application execution. The 2^{nd}, 3^{rd} and 4^{th} tasks have respectively a peak of about 10GB, 6GB and 10GB.

- *deep-learning* phase: The 5^{th} task represents the deep-learning phase. This task presents a periodic pattern with memory consumption peaks going up to 50GB. Each pattern is repeated 27 times, corresponding to the parameterization of the network tiles in SLANT-27 version.

- *post-processing* phase: The 6^{th} and 7^{th} tasks model the last phase of the application, with a memory peak to respectively 3.5GB and 10GB.

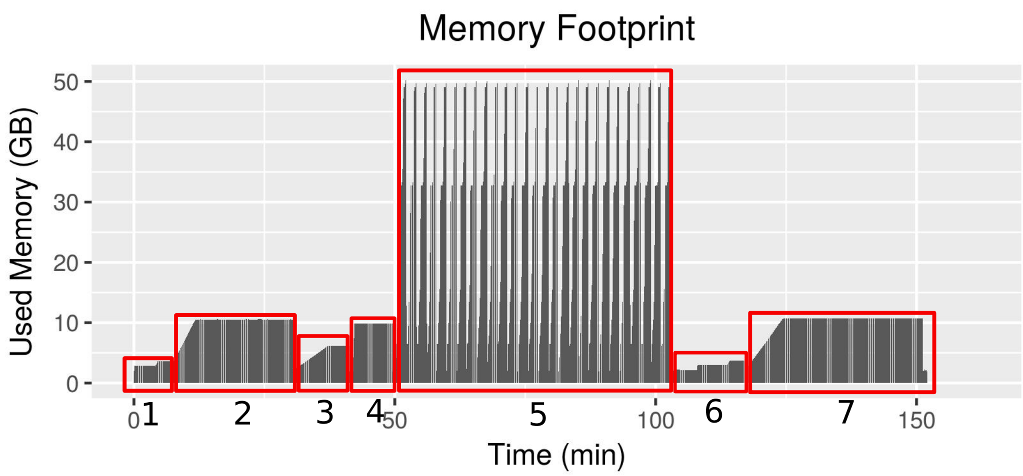


Figure 12.3: Job decomposition in tasks based on raw data of a memory footprint.

In the second step of this analysis, we are interested by the behavior of the job at the task level. For this second step, we decompose the job into tasks based on the memory characteristics by using a simple parser (see Figure 12.3). This parser returns the duration of each task within each run based on their memory footprint. Note that this decomposition can be incorrect, we discuss this and its implications later.

Using the decomposition in tasks, we can plot the individual variation of each task execution time (for simplicity, we only considered execution time at the minute level) in Figure 12.4.

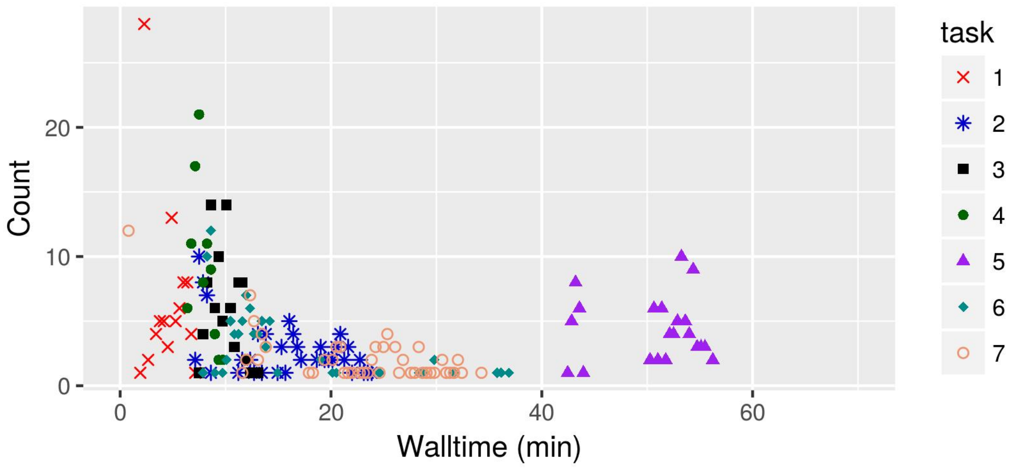


Figure 12.4: Analysis of the task walltime for all jobs (raw data).

We make the following observations. First, all tasks show variation in their walltime based on the input run. This variation differs from task to task. For instance, task #7 has variations up to ~ 25 minutes while tasks #3 and #4 have less than 5 minutes difference between runs.

Another observation from the raw data on Figure 12.4, is that some tasks present several peaks (tasks #5 and #7). There may be several explanations to this, from actual task profile (for instance a condition that adds a lot of work if it is met), lack of sufficient data for a complete profile, or finally a bad choice in our task decomposition. Going further, one may be interested in generating a finer grain parsing of the application profile to separate these peaks into individual tasks, based on more parameters than only the memory consumption. We choose not to do this to preserve some simplicity to our model. In the following, we denote by $X_1, \dots, X_7$ the random variable that represents the execution times of the seven tasks.

An important next question is whether they show correlation in their variation. Indeed, given that they are based on the same input, one may assume that they vary similarly. To study this, we present in Table 12.1 their Pearson Correlation coefficients. We see that only tasks #1 and #2 present a very high Pearson correlation (meaning that their execution times are proportional), while the others do not seem to show any impactful correlation. This measure is an important artifact as it hints at the independence of the different execution time variables.

Finally, to investigate the distribution of memory usage overtime, we study the task status at all time (at time t, which task is being executed). To do so, given X_i ($i = 1 \dots 7$) the execution time of task i, we represent in Figure 12.5 the functions $y_i(t) = \mathbb{P}\left(\sum_{j \le i} X_j < t\right)$. Essentially, it means that y_i is the probability that task i is finished.

Figure 12.5 is read this way: the probability that task i is running at time t corresponds to the distance between the plots corresponding to task $i-1$ and task i. For

Task Index	1	2	3	4	5	6	7
1	1.000	0.998	-0.308	-0.261	-0.114	-0.039	0.139
2		1.000	-0.293	-0.277	0.142	-0.058	0.159
3			1.000	0.076	0.547	-0.283	0.223
4				1.000	-0.361	0.296	-0.308
5					1.000	-0.568	0.574
6						1.000	-0.475
7							1.000

Table 12.1: Pearson Correlation matrix of the walltimes of the different tasks.

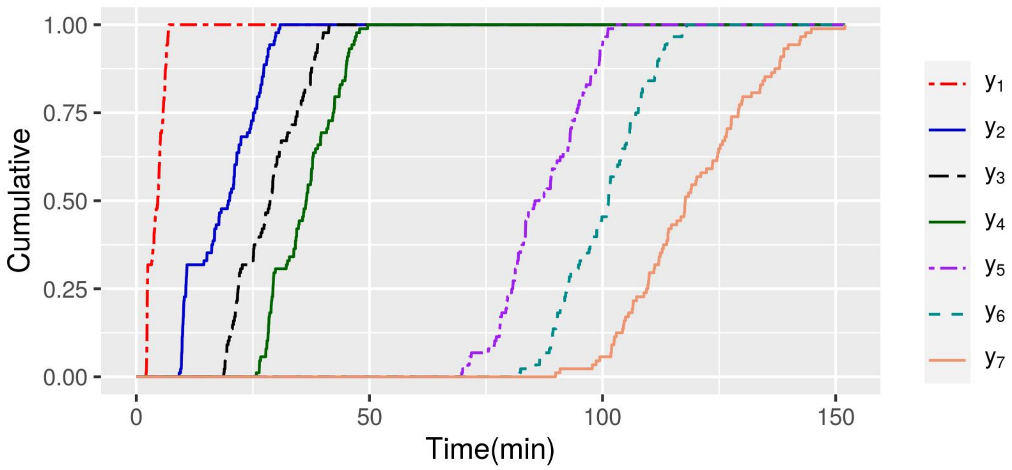


Figure 12.5: $y_i(t) = \mathbb{P}\left(\sum_{j\leq i} X_j < t\right)$ is the probability that task i is finished at time t (raw data).

instance, at time $t = 0$ task #1 is running with probability 1. At time 100, tasks #5 to #7 are running (roughly) with respective probability 0.06, 0.5, 0.38. In addition, with probability 0.06 the job has finished its execution.

This figure is interesting in the sense that it gives us task properties as a function of time. For instance, given the memory footprint of each task, one can estimate the probability of the different memory needs.

12.2 From observations to a theoretical model

Using the observations from Section 12.1, we now derive a new computational model in Section 12.2.1. We discuss the advantages and limitations of this model in Section 12.2.2.

12.2.1 Job model

We model an application A as a chain of n tasks:

$$A = j_1 \rightarrow j_2 \rightarrow \cdots \rightarrow j_n,$$

such that j_i cannot be executed until j_{i-1} is finished. Each task j_i is defined by two parameters: an execution time and a peak memory footprint. The peak memory footprint of each task does not depend on the input, and hence can be written as M_i. The execution time of each task is however input dependent, and we denote by X_i the random variable that represents the execution time of task j_i. X_i follows a probability distribution of density (PDF) f_i. We also assume that the X_i are independent.

Finally, the compact way to represent an application is

$$\{(f_1, M_1), \ldots, (f_n, M_n)\}. \tag{12.1}$$

12.2.2 Discussion

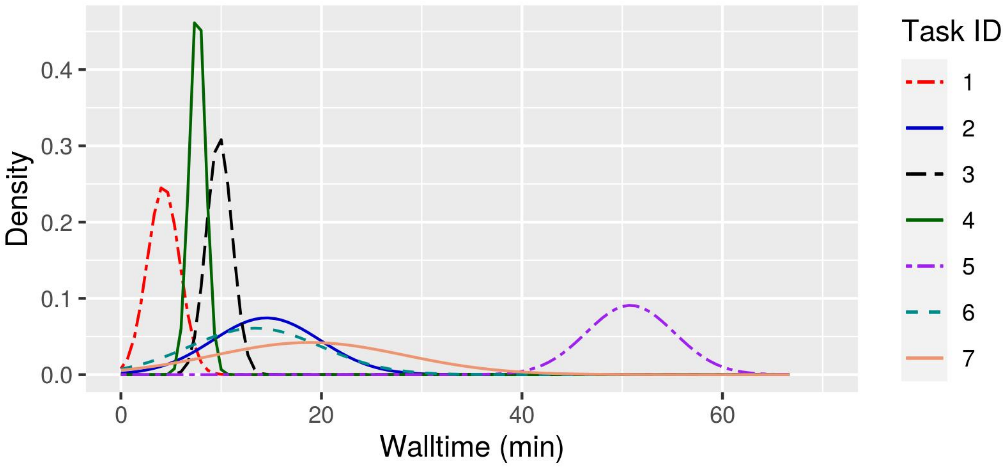


Figure 12.6: Interpolation of data from Figure 12.4 with Normal Distributions.

To discuss the model, we propose to interpolate the data from our application with Normal Distributions[1]. We present such an interpolation on Figure 12.6 (data in Table 12.2). Fitting to continuous distributions is interesting in terms of data representation, and offers more flexibility to study the properties of the application. As we have

[1]We write that X follows a normal distribution $\mathcal{N}(\mu, \sigma)$.

Task ID	1	2	3	4	5	6	7
Mean μ (in sec)	255	871	588	459	3050	804	1130
Std σ (in sec)	96.7	322	76.8	48.1	263	393	568

Table 12.2: Parameters (μ, σ) of the Normal Distributions interpolated in Figure 12.6.

seen earlier, Normal Distributions may not be the best candidates for those jobs (for example jobs with multiple peaks), but they have the advantage of being simpler to manipulate. This is also a good element to discuss the limitations of our model.

Using the interpolations, one can then compute several quantities related to the problem with more or less precision. We show how one would proceed in the following.

Task status with respect to time

We can estimate the functions $\mathbb{P}\left(\sum_{j\leq i} X_j < t\right)$ represented in Figure 12.5, which later help to guess the task status with respect to time. Indeed, if $X_1, \ldots, X_i$ are independent normal distributions of parameters $\mathcal{N}(\mu_1, \sigma_1), \ldots, \mathcal{N}(\mu_i, \sigma_i)$, then $Y_i = \sum_{j\leq i} X_j$ follows $\mathcal{N}(\sum_{j\leq i} \mu_j, \sqrt{\sum_{j\leq i} \sigma_j^2})$. We plot in Figure 12.7 the functions $f_i = \mathbb{P}(Y_i < t)$.

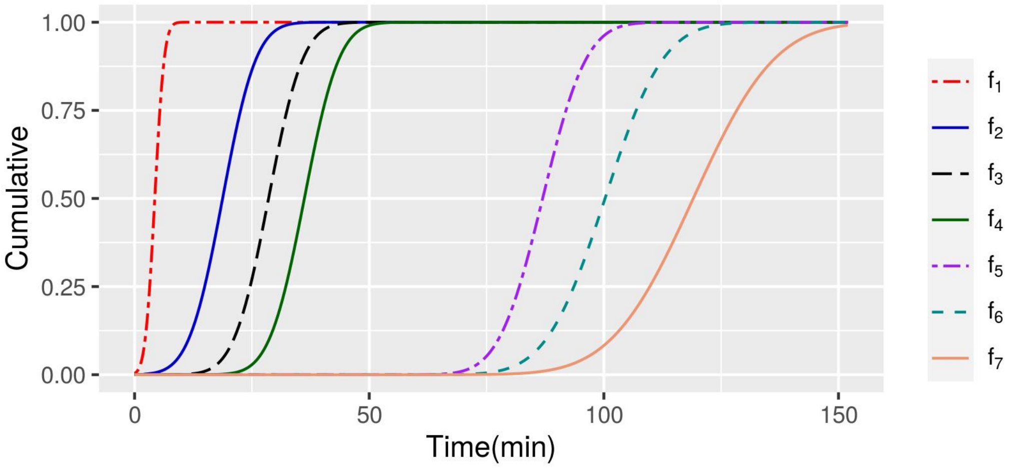


Figure 12.7: Representation of the cumulative distribution of the termination time of the 7 tasks over time from raw data.

An important observation from this figure is that even if the interpolations per task are not perfect, the sum of their model gets *closer* with time to actual data. This is further discussed in Section 12.3. Obviously this may not be true for all applications and is subject to caution, however the fact that initially all models seemed far off on a per task basis but converged well is positive.

Memory specific quantities

Using this data, one should be able to compute different metrics needed for an evaluation, such as:

- The average memory needed for a run $\bar{M} = \sum_{i=1}^{n} M_i \mathbb{E}[X_i]$. This quantity may be useful for co-scheduling schemes in the case of shared/overprovisionned resources [32, 117];
- Or even arbitrary values such as, the "likely" maximum memory needed as a function of time

$$M_\tau(t) = \max\left\{ M_i | \mathbb{P}\left(\sum_{j<i} X_i < t \leq \sum_{j \leq i} X_i \right) > \tau \right\}. \tag{12.2}$$

  We introduce this value as it will be used in Section 12.3.1.

In addition, the data for the values of M_i can be obtained with traces of very few executions (since it is not input dependent).

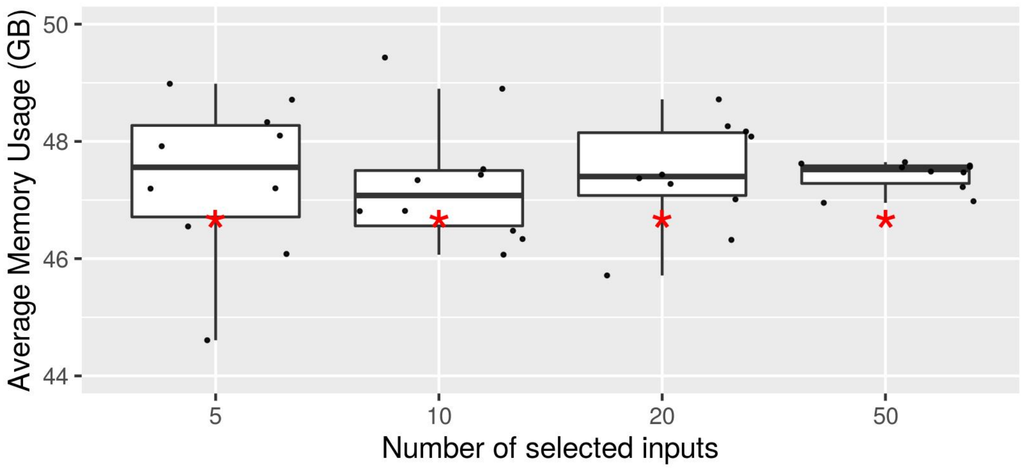
The f_i can also be interpolated from very few executions with more or less precision. We evaluate this precision here with the following experiment, presented in Figure 12.8. We interpolate from 5, 10, 20, 50 randomly selected (with replacement) runs the functions f_i and compare (i) the evolution of $\bar{M}$; and (ii) the maximum memory need $t \mapsto M_{0.1}(t)$. Each experiment is repeated 10 times to study the variations.

We observe from Figure 12.8a that with respect to the average memory need, increasing the number of data elements does not improve the precision significantly. This was expected since the only information needed is the expectation of the random variables, which is a lot easier to obtain than the distribution.

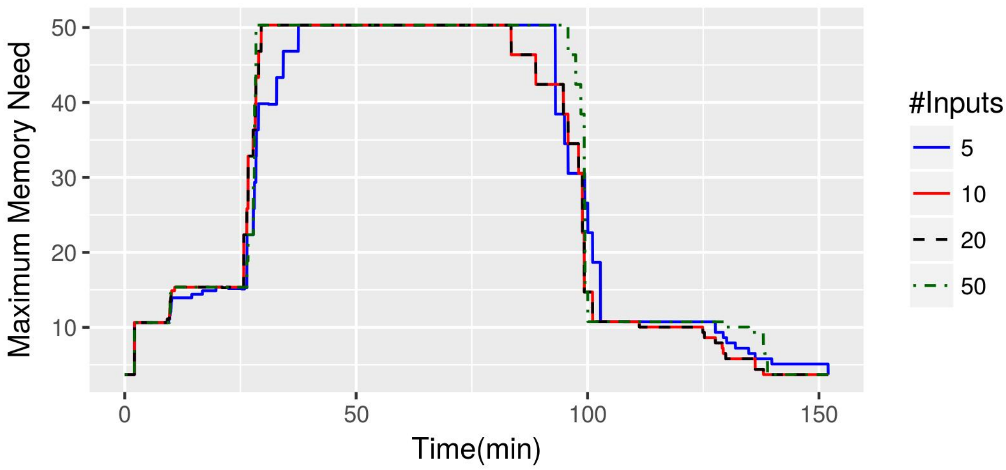
With respect to the maximum memory requirements (Figure 12.8b), it seems that very few runs (5 runs) already give good performance. This could also be predicted due to the *Maximum* function, which gives more weight to any single run.

Obviously this modelization is not perfect and can be improved depending on the level of precision one needs, specifically we can see the following caveats:

- The peak memory is different from the average memory usage (see for instance task #5 in Figure 12.3), where the job varies between high-memory needs and low-memory needs. Hence using peak memory to guess the average memory may lead to an overestimation of the average memory (as shown in Figure 12.8a). To mitigate this, one may add as a variable the average memory per task.



(a) Average memory $\bar{M}$ for different number of inputs over 10 experiments. Red star is $\bar{M}$ of original 88 runs.



(b) $M_{0.1}$ for different number of inputs (average of 10 experiments).

Figure 12.8: Different quantities that can be interpolated from the model, such as average memory (top) or peak memory (bottom).

- The model assumes that the lengths of the tasks are independent. However this may not be true as we have seen in Table 12.1 where the lengths of tasks #1 and #2 are highly correlated. In our case, a simple way to fix this would have been to merge them into a single meta task. We chose not to do this to study voluntarily the limits of the model.

- This model is based on the information available today. Specifically, the jobs here are sequentialized (the dependencies are represented by a chain of tasks). However we can expect a more general formulation where the dependencies are more parallel (and hence represented by a Directed Acyclic Graph instead of a linear chain).

To conclude this section, we have presented a new model for emerging HPC applications that is easy to manipulate but still seems close to the actual performance. We discussed possible limitations to this model. In the remaining of this chapter, we present an algorithmic use-case where one can use this model, and show on experiments that solutions derived from this model are efficient.

12.3 Impact of stochastic memory model on reservation strategies

In this section, we now discuss how our model may be used to inform on reservation strategies for HPC schedulers.

We already presented reservation strategies in Chapter 9. Essentially, for an application of unknown execution time, the strategies provided users with increasingly-long reservations to use for submission until one was sufficient to execute the whole job. We proposed strategies with optional use of checkpointing in order not to waste what was previously computed. In this chapter, we are obviously interested in the strategies with the possibility of checkpointing.

12.3.1 Algorithmic framework

In this section, we present memory-aware reservation strategies as well as the different reference heuristics we used a baseline.

Reservation strategy

A reservation strategy is presented under the form

$$\mathcal{S} = ((R_1, T_1, C_1), (R_2, T_2, C_2), \ldots, (R_n, T_n, C_n)) .$$

The strategy would then be executed as follows: initially, the user asks to the system a reservation of length $R_1+T_1+C_1$ (time to restart from previous checkpoint, the estimated walltime and the time to checkpoint at the end of the reservation). During the initial R_1 units of time, the application gathers the data needed for its computation. Then, during a time T_1 it executes. If the walltime is smaller than T_1, then the user saves the output data and the run ends. Otherwise, at the end of these T_1 units of time, the application checkpoints its current state during the C_1 units of time.

- If C_1 is enough to perform the checkpoint, then the user repeats the previous step with a reservation of length $R_2 + T_2 + C_2$.
- If C_1 is not enough to perform the checkpoint, then the user repeats the previous step with a reservation of length $R_1 + T_1 + T_2 + C_2$.

Finally, we associate to each (R_i, T_i, C_i) in $\mathcal{S}$ a memory request M_i that corresponds to an estimation of the minimum amount of memory for the application not to fail during this reservation. Typically, this value is the maximum peak of the reservation during its computation of T_i units of time. This can be obtained by tracking the progress of the application over reservations. Then, using the likely maximum memory needed as presented in Fig 12.8b, one is able to estimate the maximum memory need of the application.

Evaluated algorithms

In this work, we compare three algorithms to compute the reservation strategies. All these strategies are based from the same input: k previous runs of the application (in practice we use $k = 5,\ 10,\ 20,\ 50$).

- ALL-CHECKPOINT (Section 10.1.4 in Chapter 10): This computes the optimal solution to minimize the expected total reservation time when all reservations are checkpointed and when the checkpoint cost is constant. We take the maximum memory footprint over the execution as the basis for the checkpoint cost.
- MEM-ALL-CKPT: it is an extension of ALL-CHECKPOINT based on Section 12.2.1. Specifically it uses $M_{0.1}$ (defined in Eq. (12.2)) as the basis for the checkpoint cost function. The complete procedure of this extension is described below.
- NEURO [61, 91]: This is the algorithm used by the neuroscience department at Vanderbilt University. In their algorithm, they use the maximum length of the last k runs as their first reservation. If it is not enough, they multiply it by 1.5 and repeat the procedure. To be fair with the other strategies, we added a checkpoint to this strategy. Hence the length of the second reservation (T_2) is only 50% of the first one (T_1), so that $T_1 + T_2 = 1.5T_1$. We use the maximum size of a checkpoint as checkpoint cost. For completeness, we also add a strategy that uses average length instead of maximum length. We denote it by NEURO-AVG.

The strategies of both ALL-CHECKPOINT and MEM-ALL-CKPT assume that we have a discrete distribution of execution time for the application. Hence they start by a *modeling phase* using the k inputs. In order to do so, we fit the walltime of the k runs to a normal distribution. We then discretize it into 1000 equally spaced values on the truncated domain $[0, Q(10^{-7})]$ (where $Q(\varepsilon)$ is the ε quantile of the distribution). In addition, we then model a checkpoint cost via a simple latency/bandwidth model, where given a latency l and a bandwidth b, the checkpoint time for a volume of data V is $C(V) = l + V/b$.

After discretization, we obtain a random variable $Y \sim (v_i, C_i, f_i)_{1\leq i\leq n}$, such that for $1 \leq i \leq n, \mathbb{P}(Y = v_i) = f_i$. The cost to perform a checkpoint at time v_i is $C_i = C(M_{0.1}(v_i))$ for MEM-ALL-CKPT. We assume the cost to restart is constant R. Finally, we apply the following dynamic programming procedure to Y ($v_0 = 0$):

$$\begin{cases} S_{\text{MAC}}(n) = 0 \\ S_{\text{MAC}}(i) = \min_{i+1 \le j \le n} \left(S_{\text{MAC}}(j) + (R + (v_j - v_i) + C_i) \cdot \sum_{k=i+1}^{n} f_k \right) \end{cases}$$

MEM-ALL-CKPT and ALL-CHECKPOINT are then the associated solutions to $S_{\text{MAC}}(0)$ (depending on the checkpoint function). They can be computed in $\mathcal{O}(n^2)$ time.

12.3.2 Experimental setup

In this section, we provide information about the evaluation. The execution of the application is performed on the Haswell platform. The k inputs chosen for the modeling phase used to derive the algorithms are picked uniformly at random with replacement in the DRD set. The evaluation is performed on the set of 88 inputs from DRD. All evaluations are repeated 10 times.

Checkpointing

SLANT is currently available within a Docker image. We used the CRIU external library [120] to perform system level checkpointing of the Docker container without changing the code of SLANT. With each execution of SLANT, we are running a daemon in charge of triggering checkpoints at the times given by our strategy.

Actual checkpointing could not be used on the Haswell platform because Docker is not available there. Moreover, CRIU requires credentials on the platform that we could not get on that platform.

To workaround this problem, we use a *KNL* platform presented in Appendix D.2. This KNL platform is too slow to perform thorough experiments but Docker checkpointing is supported. Hence experiments on KNL were performed using the checkpoint times (corresponding to the right memory footprint) from that platform and simulated checkpoints (based on the KNL checkpoints) for the Haswell machine. Before doing so, we verified that the memory footprint was identical over the different phases between the two platforms. To evaluate the latency and bandwidth we use the `dd` unix command with characteristics typical for the CRIU library (multiple image files in Google protocol buffer format [2]).

Performance evaluation

Given a reservation strategy consisting of two reservations $(R_1, T_1, C_1), (R_2, T_2, C_2)$ and an application of walltime t, s.t. $T_1 < t \le T_2$, we define:

1. Its total reservation time: $(R_1 + T_1 + C_1) + (R_2 + (T_2 - T_1) + C_2)$. That is:

$$R_1 + R_2 + T_2 + C_1 + C_2;$$

2. Its system utilization, i.e. its walltime divided by its reservation time:
$$\frac{t}{R_1 + R_2 + T_2 + C_1 + C_2};$$
3. In addition, if we define M_1 and M_2 the memory requested for the reservations, we can define the weighted requested memory as:
$$\frac{(R_1 + T_1 + C_1) \cdot M_1 + (R_2 + T_2 - T_1 + C_2) \cdot M_2}{R_1 + R_2 + T_2 + C_1 + C_2}.$$
Intuitively, this is the total memory used by the different reservations normalized by time.

We present in Figure 12.9 several performance criteria to compare the different algorithms. We first discuss from a high level before entering specifics. Overall, using the improved model from Section 12.2 to design the reservation algorithm allows us to improve performance on all fronts. In addition, this model does not use much data, since performance with $k = 5$ is almost as good as performance with $k = 50$. This is an important result that shows the robustness of the model designed to the various approximations that are made (independence of variables etc).

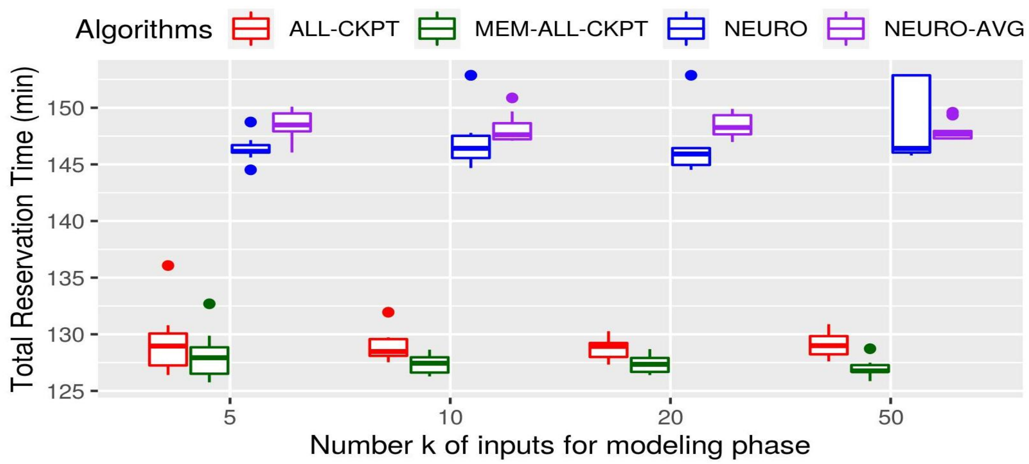
Figure 12.9a presents the results for the total reservation time metric. NEURO and NEURO-AVG have an higher reservation time, which can be expected because they are naive strategies. An interesting observation is that more data does not help it (on the contrary). This is due to the fact that with more data, the strategy includes more outliers, and since the initial reservation uses the maximum length, it guarantees an overestimation every time. MEM-ALL-CKPT performs better than all ALL-CHECKPOINT, but the difference is not large. This is probably due to a better estimation of the reservation time for the checkpoint. The observations are similar for the utilization (Figure 12.9b), for similar reasons.

Finally, Figure 12.9c plots the weighted average requested memory. ALL-CHECKPOINT and NEURO are not memory-aware, and hence assume a constant memory footprint of 51GB throughout execution. In this figure we are more interested by the performance of MEM-ALL-CKPT. The gain is $\sim 8\%$ and corresponds to the runs that needed to use a second reservation (the first one always covers task #5 and hence also has a peak memory of 51GB).

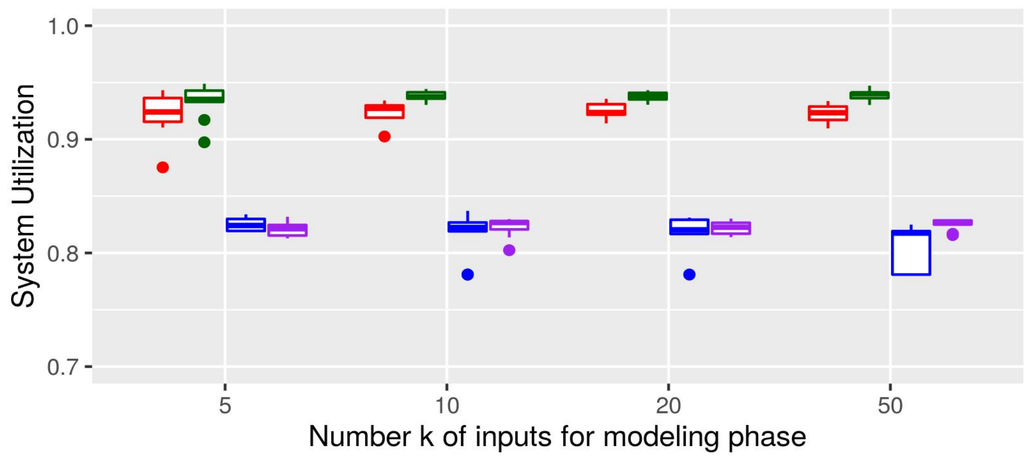
One could argue that the performance improvements between ALL-CHECKPOINT and MEM-ALL-CKPT are not that important (less than 4% in reservation time and utilization, 8% in memory usage). What is interesting here is that the only improvement we made to the ALL-CHECKPOINT strategy is the incorporation of the task-level model (and hence the memory model).

Going further

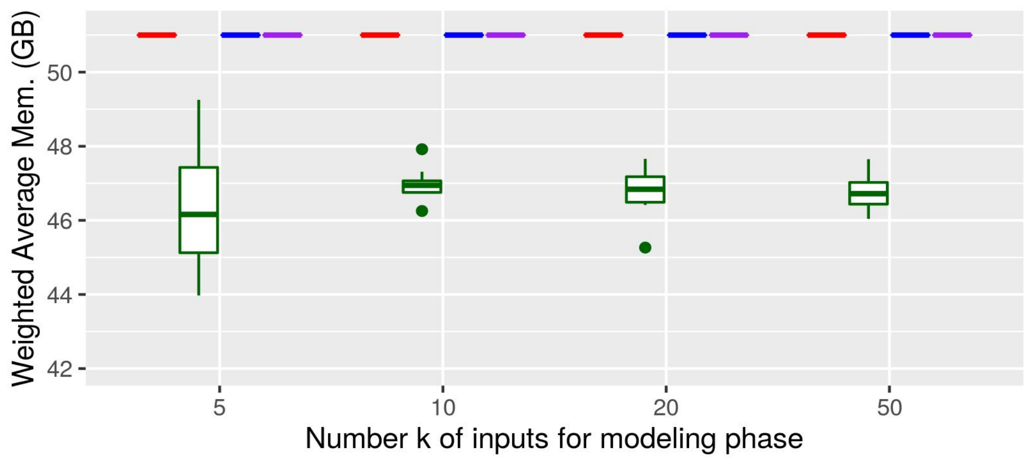
The next step would be to see how one could deduce a new and improved algorithm by using the task-level information. Specifically, looking at Figure 12.8a, the natural



(a) Average reservation time.



(b) Average utilization.



(c) Weighted average memory.

Figure 12.9: Performance of the different algorithms for various criterias.

intuition is to make a first reservation of length 25 min (guaranteed to finish before the memory intensive task #5), allowing it to be a cheaper solution memory-wise.

We study the new version of MEM-ALL-CKPT: MEM-ALL-CKPTV2 that incorporates this additional reservation. In this solution, if task #4 finishes before these 25 minutes, we cannot start task #5 since we do not have enough memory available, hence we checkpoint the output and waste the remaining time.

Figure 12.11 illustrates the scaling of the reservations between the two platforms. Remind that experiments on KNL were performed using the checkpoint times (corresponding to the right memory footprint) from that platform and simulated checkpoints (based on the KNL checkpoints) for the Haswell machine.

We plot in Figure 12.10 the total reservation time and weighted average requested memory for ALL-CHECKPOINT and MEM-ALL-CKPTV2.

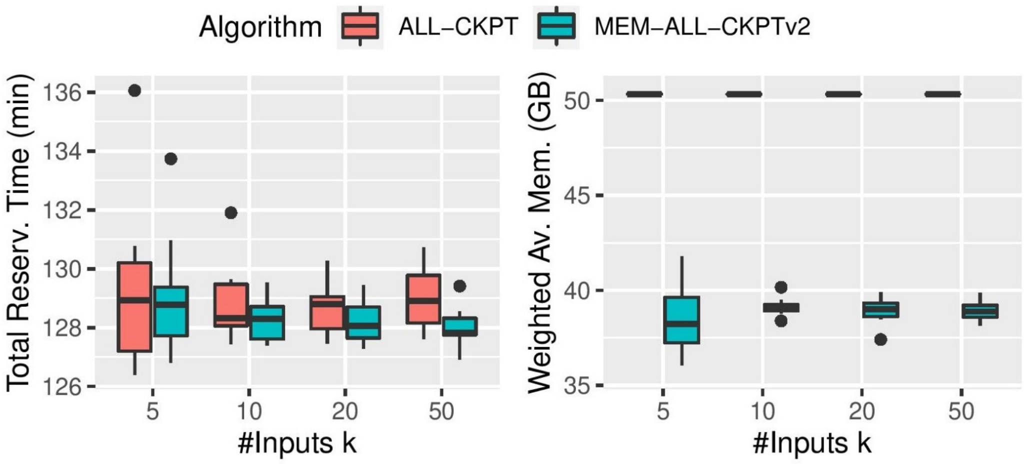
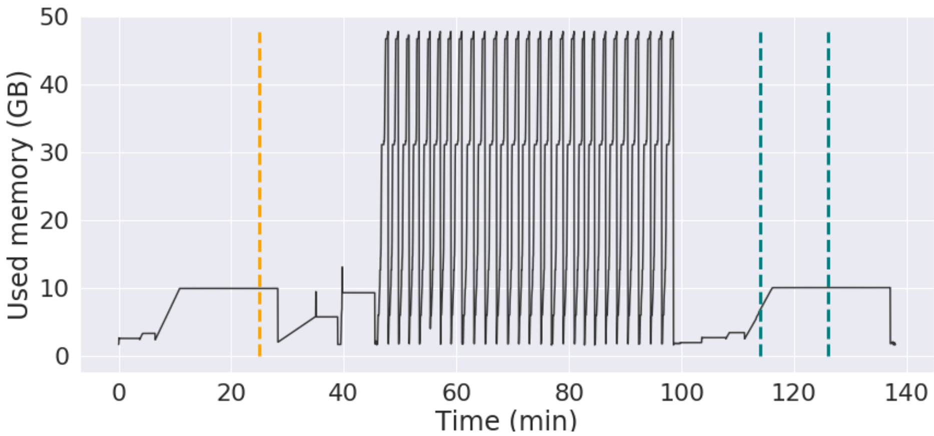
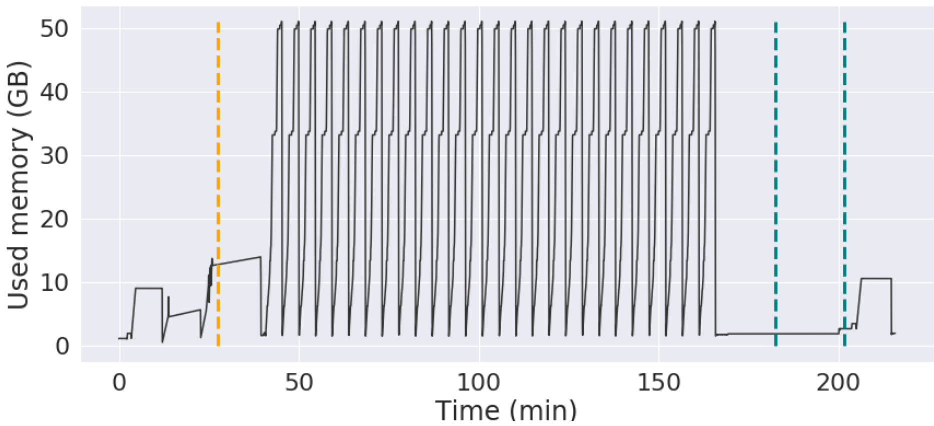


Figure 12.10: Weighted average requested memory for ALL-CHECKPOINT and MEM-ALL-CKPTV2

We see that now that MEM-ALL-CKPTV2 can gain $\sim 25\%$ of memory in average in comparison with ALL-CHECKPOINT, at no cost reservation-wise. Moreover, the cost of the extra reservation on the reservation time of the sequence is negligible. This is a very persuasive exemple of the importance of a good knowledge of application features when designing cost-efficient reservation strategies.



(a) Execution on the Haswell platform.



(b) Execution on the KNL platform.

Figure 12.11: Memory footprint of SLANT on the platforms. Vertical lines indicate the reservations given by the MEM-ALL-CKPTV2 using the Haswell platform and scaled for the KNL platform.

Chapter 13

Summary of Part II

In this chapter, we summarize all the contributions presented in the second part of this document. We also discuss short-term extensions to the solutions we proposed.

13.1 Summary

In this second part of the thesis, we presented reservation strategies for stochastic applications. By modeling the execution time of the application as a known probability distribution, we derived solutions that compute sequences of reservations that minimize the expectation of the cost of running the target application on a platform to which is associated a cost function. We considered two main application models: one that allows checkpointing some well-chosen reservations in order to save the progress during current reservation, and one that does not have any checkpointing considerations. For these two models, we derived solutions for both continuous and discrete probability distributions of the execution time of applications. Furthermore, we proposed a complete set of simulation and experiment results that demonstrate the efficiency of our strategies.

Finally, we proposed to move one step further in our study by performing an in-depth profiling of a representative stochastic application called SLANT. Our studies confirmed the variation in execution time of these applications and validated the correctness of modeling the execution time of applications with a probability distribution. Moreover, we were able to enhance our application model, by demonstrating that SLANT could be modeled with a sequence of tasks, each following its proper distribution of the execution time and having its proper memory peak. This last feature promises to greatly enhance the quality of the memory modelization of stochastic applications. For instance, in the case of SLANT, by leveraging the knowledge that task #5 has a huge memory peak in comparison with the other tasks, we were able to optimize the memory usage of reservations for which the probability of running task #5 is unlikely. Moreover, we demonstrated the efficiency of our strategies in a real-world setup. By this application profiling, we wanted to convince about the necessity to better understand application behavior and needs in order to optimize their execution on reservation-based platforms.

13.2 Perspectives

Many short-term perspectives to this work can be envisioned. Some theoretical studies still remains to be performed, such as the approximation proof for the unbounded continuous distributions in the model with checkpointing considerations (cf. Table 10.1). Designing memory-aware algorithms with performance guarantees is also a promising direction for the design of efficient strategies. Some other derivations concerning the evaluation of the critical checkpoint cost for which checkpointing all reservations becomes too costly in comparison with its pendent without checkpoint can also be envisioned.

The different models that we proposed in this part of the thesis could be extended to application running on GPUs. We expect that similar observations would be performed in comparison with the one we presented for the CPU version of applications. Besides, GPU computing is the most typical way of developing such type of applications, due to the AI techniques employed by the applications. Indeed, most AI frameworks are optimized to perform more efficient computations on GPUs rather than CPUs.

Further extensions of our proposed solutions could also include requests with variable amount of resources, which would consist in the combination of a reservation time and a number of processors. Many researches are currently dedicated to the parallelization of the AI frameworks [29, 160]. Consequently, such models would be adapted for the future enhancement of stochastic applications.

Through our study of scheduling stochastic applications, we would like to motivate users and system administrators of HPC platforms to consider adapting our strategies in the case of stochastic job submissions. In particular, by modeling the execution time of a stochastic job with a distribution, we have demonstrated the benefits of requesting a sequence of reservations compared to a single reservation of the maximum execution time for the job. Thus, by leveraging such knowledge on the job profiles, we believe that our strategies will lead to significant improvements in terms of both system and application performance over the existing scheduling policies used in HPC platforms. To do so, we would like to see the effect of such strategies on the platforms. This is a complex perspective as it requires to analyze the behavior of applications running on the platform, which necessitates to have an access to its logs and perform a fine-grain analysis of the effect of our strategies. We further develop this perspective in the conclusion of this manuscript.

Finally, we have shown that the cost of checkpoint/restart could vary, depending on where the snapshot is taken in the application execution. We started to present memory-aware solutions that leverage this variation. Further studies about the robustness to variations have to be done, due to the crucial importance of checkpoint and restart cost estimation in order to avoid the possible failure of a reservation.

Conclusion and Perspectives

Scientific simulations are compute and data-intensive programs that researchers develop in order to simulate the behavior of a natural phenomenon that is often difficult to study in laboratory conditions. Supercomputers are large-scale infrastructures that have been designed to support these very heavy programs. With the raise of computational needs in many categories of applications, supercomputers have recently been more and more employed to execute a wide range of application profiles. In this thesis, we have proposed strategies for users to optimize two different categories of HPC applications running on large-scale facilities. Part I focused on data-intensive applications that are developed to simulate complex real-life phenomena from fields such as cosmology, physics or biology. Part II was dedicated to the study of stochastic applications. Arising from emerging fields such that neuroscience, these applications display important variations in terms of execution time according to their input. On reservation-based platforms, this behavior becomes a critical issue since users need to estimate as accurately as possible the duration of their job before submitting a reservation on large-scale facilities.

Summary of contributions

The next paragraphs summarize our contributions to the issues under study. Additionally, we will discuss the prospects of this thesis and suggest a few research directions for future works that would leverage our findings.

Data-intensive applications

In Part I, we proposed models to optimize *in situ* processing for data-intensive applications. The goal is to provide data management strategies for one of the major bottlenecks hindering application performance. The *in situ* paradigm consists in coupling both simulation and analytics phases in order to post-process data directly on machine nodes, therefore avoiding the need for an intermediate storage. We expressed different mathematical models of machines and applications to represent the *in situ* pipeline between simulation and analytics from a theoretical perspective.

We then formulated a solution to optimize the *in situ* processing with the goal of minimizing the execution time of the application. We also derived solutions to this problem in the form of two complementary sub-problems: the partitioning of machine resources and the scheduling heuristics of the different tasks of the application.

Finally, we performed an evaluation of our models for synthetic applications. We pointed out the importance of memory usage during the analysis as a central feature to take into account when performing the scheduling of the different analysis functions. As a result, the amount of analysis to be performed *in situ* has to be maximized.

This work has been published in the IJHPCA journal [169].

Stochastic applications

Part II presents reservation-based strategies for stochastic applications. Stochastic applications have recently appeared on HPC facilities due to their increasing needs in terms of computations. They feature a very different profile when compared with usual HPC frameworks: their execution time is input-dependent, and cannot be predicted by an analyze of the input nature. In this part, we proposed to tackle this problem by modeling the execution time of these applications with a random variable that follows a probability distribution which is known *a priori*. For instance, the distribution can be estimated thank to the historic of application runs. From raw data, one can fit a continuous distribution, which has been demonstrated as a more expressive model than the discrete one.

We first introduced the different features of this type of applications by describing high-level observations from a representative neuroscience application. We validated the variation of execution time and demonstrated that there is no correlation between input size and execution time.

Secondly, we proposed different models for the application as well as the expression of a generic cost function for any reservation-based platform. Application models consider the possibility to take or not a checkpoint at the end of some well-chosen reservation in order to save application progress. We assumed that the memory model of the application is flat, hence that the cost of checkpoint/restart is constant all over application execution. We then formulated the optimization problems related to the two different application models.

We then exposed the algorithmic solutions to the problem under study for each considered application model, and for both discrete and continuous probability distributions. Depending on the application model, solutions are either sub-optimal heuristics, near-optimal approximations or optimal solutions. Furthermore, we proposed a complete set of evaluation results for all the aforementioned solutions to show their efficiency as well as their applicability in a real-case setup. These evaluations contain both simulation results and real-setup experiments in an HPC environment.

The last chapter of Part II carried out an in-depth profiling of a representative stochastic application. This study aims at better understanding the different features of the walltime variation. Using different observations, we refined the application model, notably by removing the flat memory model assumption. Finally, we demonstrated that a good knowledge of applications is essential to the design of cost-efficient strategies.

A first publication in IPDPS'19 [166] has introduced the application model without checkpointing, as well as the derived solutions for any probability distribution, either with continuous or discrete support. We demonstrated the benefits of our solutions in comparison with more naive strategies that do not rely on a characterization of an optimal solution. Eventually, we provided a research report [172] presenting a more complete set of results based on a various range of scenarios in order to prove the soundness of our solutions.

A second publication in IPDPS'20 [167] has developed the application model with checkpoint/restart technique with the aim of saving application progress through the different reservations. The main novelties and contributions of this work (compared to the previous one) are the novel algorithmic techniques used to provide an optimal solution for any discrete distributions, and a near-optimal solution for continuous distribution with bounded support. A set of simulation results and experiments demonstrated the efficiency and practicality of our strategies. Furthermore, a research report [173] synthesized all the results and contributions brought in the design of strategies considering checkpoint. The profiling of stochastic applications presented in Chapter 12 is proposed in [170], to appear.

Different kinds of contributions

Overall, both parts of the manuscript contain important work in terms of modeling of the optimization problem for the target applications, and propose significant algorithm development: Part I contains scheduling heuristics for the analysis functions of the applications while Part II also includes important algorithm contributions aiming to derive reservation strategies for stochastic applications. The latter also proposes substantial proofs and theoretical guarantees of the performance of these solutions.

Both parts also detail contributions in the form of simulation software to evaluate the proposed models. Appendix A elaborates on these contributions and presents artifacts that allow a full reproducibility of the results exposed in this manuscript. A set of real experiments have also been conducted to study the SLANT application, that are also presented in the appendix section.

Perspectives

Several perspectives have already been presented in the end of Part I and Part II that tended to further extend the results of this thesis. In this section, we would like to suggest future perspectives for long-term future works, which would apply to each category of applications. Eventually, we expose some future prospects that could globally concern both types of applications.

In situ processing for data-intensive applications

In Chapter 6, we mentioned short-term extensions that could contribute to enrich our models for both platform and application. In addition to these future works, further investigations can be carried out in order to make these models more expressive with regards to the competition for resources in HPC environments. For instance, one could work on the analysis that are only performed at certain iterations. This would consist in a simple way to carry out the kind of analysis that checks the integrity of data or saves a snapshot of some data at predefined iterations of simulation. Analysis could be performed either periodically, or at a specific frequency that could be irregular. In practice, this could be complex to handle, especially if one starts to consider dependencies between certain analysis functions.

As follows, another perspective is to consider a more intricate analysis model, that would focus on the dependencies between analysis functions. New scheduling policies will need to be introduced to consider the application as a task-graph model. Furthermore, it would be interesting to study models where analysis functions can adjust the parameters of the simulation depending on its progress or reached state in the target phenomenon. This would then lead to the necessity to reassess the simulation needs. Such feedback could be performed by designing dynamic models that are able to adapt to the different decisions during the application execution.

Finally, the rise of new analysis profiles such as the codes coming from AI framework have to be envisioned. Variations in execution time, as we have seen in the second part of this manuscript, will lead to a new difficulty in terms of resource allocations. We have explained that modes of operation have to be trained on a training set. However, depending on the size and nature of the training set, large variations of performance can occur. Variation of task execution time, either for analysis or simulation, would not be suitable in our current model. Indeed, our model expects that each task can be modeled by a single-core walltime that is assumed to be easy to determine. If an analysis finally sees its walltime doubled compared with the expectation, simulation has to wait for it to end, thus slowing down the overall application performance. On the contrary, if the walltime of a given job is shorter than expected, resources assigned to this analysis will have some idle time of computation. To solve these issues, one could plan to adapt the solution that we have developed for stochastic applications in the context of a classic HPC application framework. This is a difficult problem since some tasks would be stochastic, while some others would not. However, in the context of the convergence between many different domains, such analysis will probably soon be used to carry out scientific simulations, provided it gets carefully studied and developed. For instance, classification algorithms using learning techniques can be coupled with simulation to process its data.

Stochastic applications

In Chapter 13, we described some direct perspectives to our contributions. They mostly consist in wrapping up the remaining theoretical proofs to be derived, in addition to validating our work on the GPU version of applications. Also, we proposed to extend our strategies by considering requests with variable amount of resources as a promising direction for the enhancement of distributed AI frameworks. We now briefly present mid to long-term prospects of extension for this work.

An interesting future work for stochastic applications would be to include system constraints in our models. For instance, I/O congestion may have an important impact on reservations that perform a checkpoint. Indeed, a checkpoint consists in saving a snapshot of the execution on the PFS, and its cost is often greatly impacted by congestion. Though we have estimated what would be the time for doing a checkpoint in our strategies, we would like to point out that an under-estimation would lead to the failure of the checkpoint, which would have a direct impact on the remaining reservations that would finally not have the snapshot in memory to restart from. A solution to alleviate the I/O congestion would be to perform a checkpoint not only at the end of reservations, but also anytime during its execution. This would add more flexibility and allow us to make a checkpoint with more chances of success. Moreover, the application could itself checkpoint at a moment that is beneficial in the context of the execution, for instance between different phases. However, snapshots of applications taken at different time will not represent the same state of the application. Moreover, if a checkpoint is performed before the end of a reservation, one will have to decide what to do with the possibly remaining time once the checkpoint is over. One could continue the application process and hope that it will let the application achieve its termination. If it does not, this extra work will be lost. One should stress that this approach is not always possible: for example, it will not work for the checkpointing library CRIU that systematically kills the application after a checkpoint is performed. Another possibility would be to actually end the application after each checkpoint and move to the next reservation.

Another meaningful way to develop our findings is to include all our solutions into a more global stochastic framework. Indeed, applications are often connected to each other in a global framework. When a first application computes a result, a second one then uses them to perform another computation, and so on. This represents quite a tricky challenge because it means that the model that will be used must be elaborate enough to include several successive stochastic processes.

Ultimately, we also believe that the application's behavioral model can help understand the needs of these applications and can guide the design of future middleware for HPC systems, including I/O and memory management. Hence, pursuing the strain towards an accurate modeling of the applications appears to be an important prospect for future works.

Global perspectives

Throughout the present work, we have provided different strategies to manage different types of applications. Our approach was to mathematically represent the behavior of applications running on a target platform. We demonstrated that this approach can lead to important gain in terms of application performance, allowing users to reduce their operating expenses.

On the use of theoretical models

A natural way to pursue our work is to determine how our solutions can be used in practice. Different options can be envisioned, depending on the type of applications that will be considered.

For data-intensive applications, the most likely option would be to let users be responsible for optimizing their requests on a target machine. This way, they would have to instantiate the models corresponding to the profile of the application in question. This requires a human mediation to find the good parameters for the models, especially for users that are not used to manipulate such mathematical tools. They would have to profile the simulation phase as well as the analysis functions in order to determine the resource partitioning between simulation and analytics. In addition, users would be able to schedule the analysis functions either on helper cores or on staging nodes.

For stochastic applications, the same principle could be applied. By performing some runs of the applications, one can use the historic data to instantiate the model. Recent works have shown the benefits of using continuous models for few data, which makes our solutions totally suitable in that case. The limitations of this approach dwell in the amount of human work that it requires. Application developers and users come from different domains, and it is not necessarily easy for all of them to understand the benefits of our strategies as well as how to put them in practice. While some automatic modules could help users perform the different operations, users would still rely on basic strategies by convenience, even if we proved that they are very costly for both the users and the platforms.

Another approach could be envisioned that would improve the quality of the submissions and limit the necessary work to be performed by users. Indeed, one could assume that the scheduler could be in charge of managing the different reservations. If the scheduler is able to intercept submissions of stochastic applications, it could automatically compute a sequence of reservations for the submitted application. This would be done by accessing a database storing the past runs of the same application, which is already a challenging functionality to set up. For instance, the Slurm scheduler could deploy a special module in order to offer such solutions. The major difficulties ensuing from this idea are two-fold. The first one is to convince both users and system administrators of the benefits of such an approach. The second one is to allow the scheduler to access the necessary data in order to perform this process. The user identifier, as well as program name, should always be analyzed by the scheduler in order to detect the concerned application and automate the processing of stochastic requests.

Before deploying such process, an in-depth analysis of the history of submissions over large-scale platforms has to be performed. Accordingly, in order to optimize the processing of job submissions, a salient idea would be to analyze traces from supercomputers. Indeed, statistics about each submission dropped on a platform are registered by the system, generating an important database. For instance, various elements such as the users' identifiers, the size of each submission, the waiting time in the scheduler queue, and many others are stored in a large database. Altogether, an in-depth analysis of these data would help system administrators to better understand user behavior and to categorize jobs with regards to many different metrics. For stochastic jobs, this would allow us to determine which users or groups of users use such type of application. Then, the system could follow a specific allocation policy for the related submissions. From a more global perspective, a study of such dataset would greatly help administrators identify the limits of current scheduling and resource partitioning policies. The identified weaknesses could then be solved by providing adapted solutions that would target either specific categories of applications or the whole set of job submissions.

From static to dynamic decisions

One last possible way to relevantly carry on our results concerns our approach of the design of static strategies. Every model is instantiated once and for all at the beginning of the application run. The main drawback of such a method is that it cannot take benefit from any information during the application run. In order to resolve this negative point, dynamic models could be designed that would take advantage of the feedback concerning specific metrics of the application. For instance, the *in situ* paradigm could adapt the resource assignation to analytics depending on its performance on previous iterations. This could be very helpful if the first estimation of the resource needs turns out not to be accurate. The main difficulties of this approach in terms of model consist in determining how often the model should be refreshed, without impacting the performance of the application. In the case of stochastic applications, this approach could be quite advantageous in situations where one reservation failed during its execution, either because of a checkpoint that did not have enough time to be performed, or due to hardware failure (or other external causes). In both cases, the remaining reservations may not perform as expected. A new computation of the sequence of reservations would definitely overcome the negative impact of the failed reservation. However, depending on the considered strategies, this computation could prove to be quite time consuming. Once again, the frequency of this revaluation would need to be carefully determined to limit its impact on the overall performance. Finally, some scheduler features such as dynamic reservations could be further explored. The most crucial challenge would be to ensure that the scheduler extends current reservation before it reaches its end in order to avoid computation termination. Such an approach would need to be experimented in practice so as to evaluate how it could be integrated to the numerous strategies that we have proposed in this thesis.

Appendix

Appendix A

Software production and reproducibility

A.1 Software resource for *in situ* modeling

The software simulator of *in situ* processing for data-intensive applications is designed to evaluate resource partitioning solutions as well as scheduling heuristics for simulation and analytics of HPC simulations. It has been developed using SageMath toolkit. It implements a platform and application model. Each model has been implemented to make it as flexible as possible for users. Hence, users can parametrized at their convenience the different features related to the application and the platform.

The code is freely accessible online at `https://gitlab.inria.fr/vhonore/in-situ_simulator`. Plotting scripts and data are also available to reproduce the simulation results presented in Chapter 5.

A.2 Software resource for stochastic evaluation

We developed different software resources for evaluating the different reservation-based strategies presented in Chapter 11. All the simulations presented in this section have been performed on the Haswell platform presented in Appendix D.

A.2.1 Simulation code for stochastic evaluation without checkpoint

The first software production concerns the solutions without checkpointing consideration. We developed a simulation framework that allows to compute the expected costs of different strategies when the application execution time follows any instantiation of usual probability distributions presented in Table 11.1. The simulation code is divided in several files, each implementing either the different operations related to the probability distributions, the algorithms that compute the sequence of reservations, the cost function and computation of expected cost, or the main file that contains all the instructions to perform the computations.

All the code, setup and instructions to reproduce the experiments presented in Section 11.1 of Chapter 11 are publicly available on https://gitlab.inria.fr/vhonore/ipdps_2019_stochastic-scheduling.

A.2.2 Simulation code for stochastic evaluation with checkpoint

In order to evaluate the solutions with checkpointing, we refined the first software production of Appendix A.2.1 to include the checkpointing considerations. This implies the implementation of the whole new strategies, as well as the refinement of the computation of the expected cost for a given sequence of reservations. Moreover, different evaluations have been designed to evaluate the benefits and possible associated limitations of checkpointing in our models.

Here again, a full description of the procedures for a fully reproducibility of results presented in Section 11.2 of Chapter 11 are publicly available on https://gitlab.inria.fr/vhonore/ckpt-for-stochastic-scheduling.

A.3 Software resource for SLANT profiling

The last development in terms of code production concerns the profiling of the SLANT application, presented in Chapters 7 and 12. We executed the application on the Haswell platform presented in Appendix D.1. The application is given under the format of a Docker image, that we run on the Haswell platform using Singularity software. For each run, we monitored the memory consumption of the application and measured its execution time. This process generates a database of runs that we used to investigate the profile of such category of applications. The full procedure to run the application as well as all the generated data are publicly available at https://gitlab.inria.fr/vhonore/sc20_reproducibility_initiative.

Later on, experiments considering checkpointing with the SLANT application have been performed on a *Knights Landing* platform presented in Appendix D.2. An other repository contains our approach to run SLANT with a checkpointing library called CRIU. Overall reproducibility features are accessible at https://github.com/anagainaru/ReproducibilityInitiative/tree/master/2020_tpds. This repository is maintained by one of the collaborators to this work.

Appendix B

Recursive formulas to get sequence of reservations for MEAN-BY-MEAN heuristic

In this section, we present recursive formulas to compute the sequence of reservations $S = (t_1, t_2, \ldots, t_i, t_{i+1}, \ldots)$using the MEAN-BY-MEAN heuristic for the considered distributions. As described in Section 11.1.1, the heuristic computes a new reservation value t_i from the previous value t_{i-1} as follows:

$$t_i = \mathbb{E}(X|X > t_{i-1}) = \frac{\int_{t_{i-1}}^{\infty} tf(t)dt}{1 - F(t_{i-1})}, \text{ for all } i \geq 2 \tag{B.1}$$

The following subsections present the derivations of the formulas for different distributions, while Table B.1 summarizes results for all distributions.

B.1 Exponential distribution

For Exponential (λ), substituting $f(t) = \lambda e^{-\lambda t}$ and $F(t) = 1 - e^{-\lambda t}$ into Equation (B.1), we get:

$$\begin{aligned}
t_i &= \frac{\int_{t_{i-1}}^{\infty} t\lambda e^{-\lambda t}dt}{e^{-\lambda t_{i-1}}} \\
&= \frac{[-te^{-\lambda t}]_{t_{i-1}}^{\infty} + \int_{t_{i-1}}^{\infty} e^{-\lambda t}dt}{e^{-\lambda t_{i-1}}} \qquad \text{(integrating by parts)} \\
&= \frac{t_{i-1}e^{-\lambda t_{i-1}} + [-\frac{1}{\lambda}e^{-\lambda t}]_{t_{i-1}}^{\infty}}{e^{-\lambda t_{i-1}}} \\
&= \frac{t_{i-1}e^{-\lambda t_{i-1}} + \frac{1}{\lambda}e^{-\lambda t_{i-1}}}{e^{-\lambda t_{i-1}}} \\
&= t_{i-1} + \frac{1}{\lambda}
\end{aligned}$$

Distribution	**Sequence of t_i's for $i \geq 1$**
Exponential(λ)	$t_i = \begin{cases} \frac{1}{\lambda}, & \text{if } i = 1 \\ t_{i-1} + \frac{1}{\lambda}, & \text{otherwise} \end{cases}$
Weibull(λ, κ)	$t_i = \begin{cases} \lambda\Gamma\left(1 + \frac{1}{\kappa}\right), & \text{if } i = 1 \\ \lambda e^{\left(\frac{t_{i-1}}{\lambda}\right)^\kappa} \Gamma\left(1 + \frac{1}{\kappa}, \left(\frac{t_{i-1}}{\lambda}\right)^\kappa\right), & \text{otherwise} \end{cases}$
Gamma(α, β)	$t_i = \begin{cases} \frac{\alpha}{\beta}, & \text{if } i = 1 \\ \frac{1}{\beta}\left(\alpha + \frac{(\beta t_{i-1})^\alpha e^{-\beta t_{i-1}}}{\Gamma(\alpha, \beta t_{i-1})}\right), & \text{otherwise} \end{cases}$
LogNormal(μ, σ^2)	$t_i = \begin{cases} e^{\mu + \frac{\sigma^2}{2}}, & \text{if } i = 1 \\ e^{\mu + \frac{\sigma^2}{2}} \cdot \frac{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}\right)}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)}, & \text{otherwise} \end{cases}$
TruncatedNormal(μ, σ^2, a)	$t_i = \begin{cases} \mu + \sigma\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{a-\mu}{\sigma}\right)^2}}{1 - \text{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)}, & \text{if } i = 1 \\ \mu + \sigma\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{t_{i-1}-\mu}{\sigma}\right)^2}}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)}, & \text{otherwise} \end{cases}$
Pareto(ν, α)	$t_i = \begin{cases} \frac{\alpha}{\alpha-1}\nu, & \text{if } i = 1 \\ \frac{\alpha}{\alpha-1}t_{i-1}, & \text{otherwise} \end{cases}$ (for $\alpha > 1$)
Uniform(a, b)	$t_i = \begin{cases} \frac{1}{2}(a + b), & \text{if } i = 1 \\ \frac{1}{2}(t_{i-1} + b), & \text{otherwise} \end{cases}$
Beta(α, β)	$t_i = \begin{cases} \frac{\alpha}{\alpha+\beta}, & \text{if } i = 1 \\ \frac{\text{B}(\alpha+1,\beta) - \text{B}(t_{i-1};\alpha+1,\beta)}{\text{B}(\alpha,\beta) - \text{B}(t_{i-1};\alpha,\beta)}, & \text{otherwise} \end{cases}$
BoundedPareto(L, H, α)	$t_i = \begin{cases} \frac{\alpha}{\alpha-1} \cdot \frac{H^{1-\alpha} - L^{1-\alpha}}{H^{-\alpha} - L^{-\alpha}}, & \text{if } i = 1 \\ \frac{\alpha}{\alpha-1} \cdot \frac{H^{1-\alpha} - t_{i-1}^{1-\alpha}}{H^{-\alpha} - t_{i-1}^{-\alpha}}, & \text{otherwise} \end{cases}$ (for $\alpha > 1$)

Table B.1: Recursive formulas to compute sequence of reservations for MEAN-BY-MEAN.

B.2 Weibull distribution

For Weibull(λ, κ), substituting $f(t) = \frac{\kappa}{\lambda}\left(\frac{t}{\lambda}\right)^{\kappa-1} e^{-\left(\frac{t}{\lambda}\right)^\kappa}$ and $F(t) = 1 - e^{-\left(\frac{t}{\lambda}\right)^\kappa}$ into Equation (B.1) and simplifying, we get:

$$\begin{aligned} t_i &= \frac{\int_{t_{i-1}}^{\infty} \kappa \left(\frac{t}{\lambda}\right)^\kappa e^{-\left(\frac{t}{\lambda}\right)^\kappa} dt}{e^{-\left(\frac{t_{i-1}}{\lambda}\right)^\kappa}} \\ &= \frac{\int_{\left(\frac{t_{i-1}}{\lambda}\right)^\kappa}^{\infty} \lambda x^{\frac{1}{\kappa}} e^{-x} dx}{e^{-\left(\frac{t_{i-1}}{\lambda}\right)^\kappa}} \qquad \left(\text{by letting } x = \left(\frac{t}{\lambda}\right)^\kappa\right) \\ &= \lambda e^{\left(\frac{t_{i-1}}{\lambda}\right)^\kappa} \Gamma\left(1 + \frac{1}{\kappa}, \left(\frac{t_{i-1}}{\lambda}\right)^\kappa\right) \end{aligned}$$

where $\Gamma(x, y) = \int_y^\infty t^{x-1} e^{-t} dt$ denotes the upper incomplete gamma function.

B.3 Gamma distribution

For Gamma(α, β), substituting $f(t) = \frac{\beta^\alpha}{\Gamma(\alpha)} t^{\alpha-1} e^{-\beta t}$ and $F(t) = 1 - \frac{\Gamma(\alpha, \beta t)}{\Gamma(\alpha)}$ into Equation (B.1) and simplifying, we get:

$$\begin{aligned}
t_i &= \frac{\int_{t_{i-1}}^{\infty} \beta^\alpha t^\alpha e^{-\beta t} dt}{\Gamma(\alpha, \beta t_{i-1})} \\
&= \frac{1}{\beta} \cdot \frac{\int_{\beta t_{i-1}}^{\infty} x^\alpha e^{-x} dx}{\Gamma(\alpha, \beta t_{i-1})} \qquad \text{(by letting } x = \beta t\text{)} \\
&= \frac{1}{\beta} \cdot \frac{\alpha \int_{\beta t_{i-1}}^{\infty} x^{\alpha-1} e^{-x} dx - [x^\alpha e^{-x}]_{\beta t_{i-1}}^{\infty}}{\Gamma(\alpha, \beta t_{i-1})} \qquad \text{(integrating by parts)} \\
&= \frac{1}{\beta} \cdot \frac{\alpha \Gamma(\alpha, \beta t_{i-1}) + (\beta t_{i-1})^\alpha e^{-\beta t_{i-1}}}{\Gamma(\alpha, \beta t_{i-1})} \\
&= \frac{1}{\beta} \left(\alpha + \frac{(\beta t_{i-1})^\alpha e^{-\beta t_{i-1}}}{\Gamma(\alpha, \beta t_{i-1})} \right)
\end{aligned}$$

where $\Gamma(x, y) = \int_y^\infty t^{x-1} e^{-t} dt$ denotes the upper incomplete gamma function.

B.4 LogNormal distribution

For LogNormal(μ, σ^2), substituting $f(t) = \frac{1}{t\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\ln t - \mu}{\sigma}\right)^2}$ and $F(t) = \frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{\ln t - \mu}{\sqrt{2}\sigma}\right)$ into Equation (B.1) and simplifying, we get:

$$\begin{aligned}
t_i &= \frac{1}{\sigma}\sqrt{\frac{2}{\pi}} \cdot \frac{\int_{t_{i-1}}^{\infty} e^{-\frac{1}{2}\left(\frac{\ln t - \mu}{\sigma}\right)^2} dt}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)} \\
&= e^{\mu + \frac{\sigma^2}{2}} \cdot \frac{\frac{2}{\sqrt{\pi}} \int_{\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}}^{\infty} e^{-x^2} dx}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)} \qquad \left(\text{by letting } x = \frac{\ln t - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}\right) \\
&= e^{\mu + \frac{\sigma^2}{2}} \cdot \frac{\frac{2}{\sqrt{\pi}} \int_0^\infty e^{-x^2} dx - \frac{2}{\sqrt{\pi}} \int_0^{\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}} e^{-x^2} dx}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)} \\
&= e^{\mu + \frac{\sigma^2}{2}} \cdot \frac{\text{erf}(\infty) - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}\right)}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)} \\
&= e^{\mu + \frac{\sigma^2}{2}} \cdot \frac{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}} - \frac{\sigma}{\sqrt{2}}\right)}{1 - \text{erf}\left(\frac{\ln t_{i-1} - \mu}{\sigma\sqrt{2}}\right)}
\end{aligned}$$

where $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ denotes the error function.

B.5 TruncatedNormal distribution

For TruncatedNormal(μ, σ^2, a), substituting $f(t) = \frac{1}{\sigma}\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2}}{1-\text{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)}$ and $F(t) = \frac{\text{erf}\left(\frac{t-\mu}{\sigma\sqrt{2}}\right)-\text{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)}{1-\text{erf}\left(\frac{a-\mu}{\sigma\sqrt{2}}\right)}$ into Equation (B.1) and simplifying, we get:

$$
\begin{aligned}
t_i &= \frac{1}{\sigma}\sqrt{\frac{2}{\pi}} \cdot \frac{\int_{t_{i-1}}^{\infty} t e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2} dt}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)} \\
&= \frac{2}{\sqrt{\pi}} \cdot \frac{\int_{\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}}^{\infty} (x\sigma\sqrt{2} + \mu) e^{-x^2} dx}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)} \qquad \left(\text{by letting } x = \frac{t-\mu}{\sigma\sqrt{2}}\right) \\
&= \frac{2}{\sqrt{\pi}} \cdot \frac{\sigma\sqrt{2}\int_{\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}}^{\infty} x e^{-x^2} dx + \mu\left(\int_0^{\infty} e^{-x^2} dx - \int_0^{\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}} e^{-x^2} dx\right)}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)} \\
&= \frac{2}{\sqrt{\pi}} \cdot \frac{\sigma\sqrt{2}\left[-\frac{1}{2}e^{-x^2}\right]_{\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}}^{\infty} + \mu\frac{\sqrt{\pi}}{2}\left(\text{erf}(\infty) - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)\right)}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)} \\
&= \frac{2}{\sqrt{\pi}} \cdot \frac{\sigma\sqrt{2}\left(0 + \frac{1}{2}e^{-\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)^2}\right) + \mu\frac{\sqrt{\pi}}{2}\left(1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)\right)}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)} \\
&= \mu + \sigma\sqrt{\frac{2}{\pi}} \cdot \frac{e^{-\frac{1}{2}\left(\frac{t_{i-1}-\mu}{\sigma}\right)^2}}{1 - \text{erf}\left(\frac{t_{i-1}-\mu}{\sigma\sqrt{2}}\right)}
\end{aligned}
$$

where $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ denotes the error function.

B.6 Pareto distribution

For Pareto(ν, α) with $\alpha > 1$, substituting $f(t) = \frac{\alpha \nu^\alpha}{t^{\alpha+1}}$ and $F(t) = 1 - \left(\frac{\nu}{t}\right)^\alpha$ into Equation (B.1) and simplifying, we get:

$$
\begin{aligned}
t_i &= \frac{\alpha \int_{t_{i-1}}^{\infty} t^{-\alpha} dt}{t_{i-1}^{-\alpha}} \\
&= \frac{\alpha}{1-\alpha} \cdot \frac{\left[t^{1-\alpha}\right]_{t_{i-1}}^{\infty}}{t_{i-1}^{-\alpha}} \\
&= \frac{\alpha}{1-\alpha} \cdot \frac{0 - t_{i-1}^{1-\alpha}}{t_{i-1}^{-\alpha}} \\
&= \frac{\alpha}{\alpha - 1} t_{i-1}
\end{aligned}
$$

B.7 Uniform distribution

For Uniform(a, b), substituting $f(t) = \frac{1}{b-a}$ and $F(t) = \frac{t-a}{b-a}$ into Equation (B.1), we get:

$$
\begin{aligned}
t_i &= \frac{\int_{t_{i-1}}^{b} t \frac{1}{b-a} dt}{1 - \frac{t_{i-1}-a}{b-a}} \\
&= \frac{b^2 - t_{i-1}^2}{2(b - t_{i-1})} \\
&= \frac{b + t_{i-1}}{2}
\end{aligned}
$$

B.8 Beta distribution

For Beta(α, β), substituting $f(t) = \frac{t^{\alpha-1}(1-t)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}$ and $F(t) = \frac{\mathrm{B}(t;\alpha,\beta)}{\mathrm{B}(\alpha,\beta)}$ into Equation (B.1) and simplifying, we get:

$$
\begin{aligned}
t_i &= \frac{\int_{t_{i-1}}^{1} t^\alpha (1-t)^{\beta-1} dt}{\mathrm{B}(\alpha,\beta) - \mathrm{B}(t_{i-1};\alpha,\beta)} \\
&= \frac{\int_0^1 t^\alpha (1-t)^{\beta-1} dt - \int_0^{t_{i-1}} t^\alpha (1-t)^{\beta-1} dt}{\mathrm{B}(\alpha,\beta) - \mathrm{B}(t_{i-1};\alpha,\beta)} \\
&= \frac{\mathrm{B}(\alpha+1,\beta) - \mathrm{B}(t_{i-1};\alpha+1,\beta)}{\mathrm{B}(\alpha,\beta) - \mathrm{B}(t_{i-1};\alpha,\beta)}
\end{aligned}
$$

where $\mathrm{B}(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1} dt$ represents the beta function and $\mathrm{B}(a;x,y) = \int_0^a t^{x-1}(1-t)^{y-1} dt$ represents the incomplete beta function.

B.9 BoundedPareto distribution

For BoundedPareto(L, H, α), substituting $f(t) = \frac{\alpha H^\alpha L^\alpha t^{-\alpha-1}}{H^\alpha - L^\alpha}$ and $F(t) = \frac{H^\alpha(1-L^\alpha t^{-\alpha})}{H^\alpha - L^\alpha}$ into Equation (B.1) and simplifying, we get:

$$\begin{aligned} t_i &= \frac{\alpha H^\alpha \int_{t_{i-1}}^{H} t^{-\alpha} dt}{H^\alpha t_{i-1}^{-\alpha} - 1} \\ &= \frac{\alpha}{1-\alpha} \cdot \frac{H^\alpha (H^{1-\alpha} - t_{i-1}^{1-\alpha})}{H^\alpha t_{i-1}^{-\alpha} - 1} \\ &= \frac{\alpha}{\alpha - 1} \cdot \frac{H^{1-\alpha} - t_{i-1}^{1-\alpha}}{H^{-\alpha} - t_{i-1}^{-\alpha}} \end{aligned}$$

Appendix C

Evaluation of strategies with checkpointing using HPC cost function

In this appendix, we propose to complete the evaluation of strategies with checkpointing by providing results for HPC cost function. In compared with RESERVATIONONLY cost function, it considers an additional cost that is proportional to the actual execution time (pay for what you use). Thus, $\alpha = 1, \beta = 1, \gamma = 0$.

C.1 Results for Scenario 1

We present first an evaluation of the performance of DYN-PROG-COUNT, compared to the other strategies, when the values of R and C varies. Figure C.1 shows the results with HPC cost function. We see that results are consistent with Figure 11.3, with small variations in results but same general trends.

We then study the impact of ε on the performance of DYN-PROG-COUNT (DPC) when $R = C =$ 6min, 30min and 60min. When $\varepsilon = 1$, this theoretically guarantees that the performance is at most twice ($= 1 + \varepsilon$) that of the optimal, but in practice it can be a lot better. We study in

Figure C.2 shows the performance of DYN-PROG-COUNT for various values of ε for distributions of Table 11.1 with HPC cost function. All performance are normalized by DYN-PROG-COUNT for $\varepsilon = 0.1$. One can note that all distributions converge even faster than the results presented in Figure 11.4, for RESERVATIONONLY cost function. Indeed, for $\varepsilon = 0.4$, all distribution converge. Overall, the results are consistent in between the two cost functions. In both figures, the number of chunks n in DYN-PROG-COUNT varies between 50 to 1000 depending on the distribution and value of ε, showing the practicality of DYN-PROG-COUNT for considered distributions.

The final evaluation for this scenario is a study of the impact of the size of the period. Until now we have always chosen the period that minimized the objective functions. Table C.1 (resp. Table C.2) shows the performance of both variants of the periodic algorithms, ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized

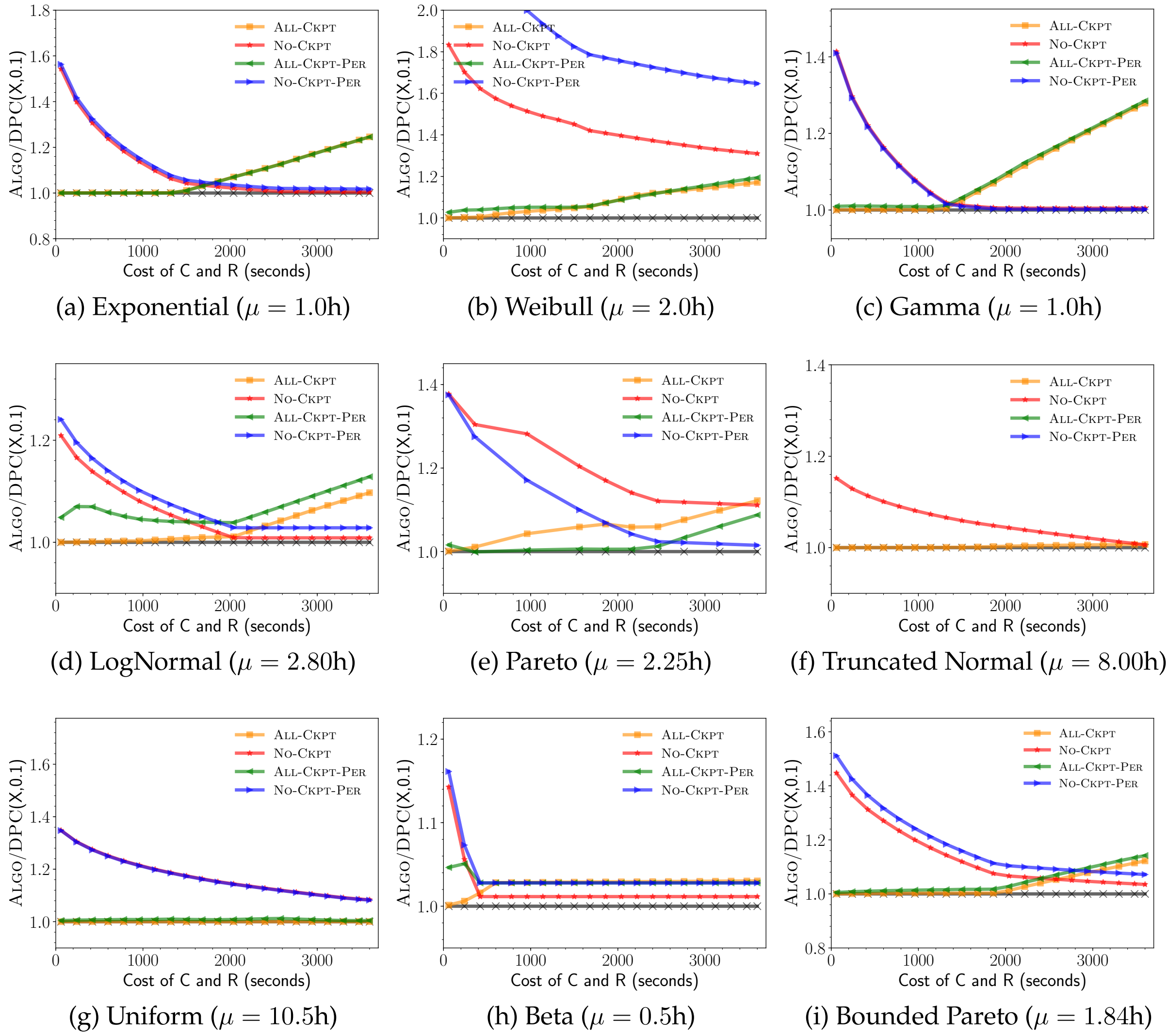
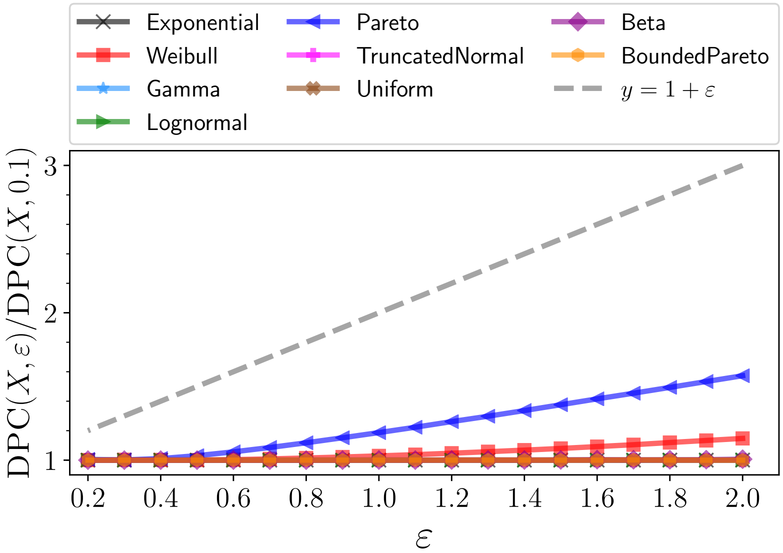
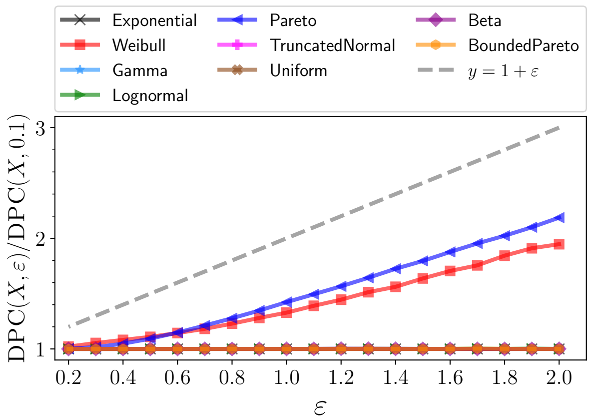


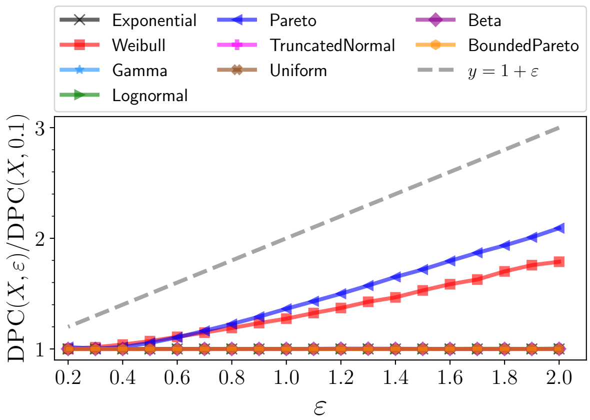
Figure C.1: Expected costs of the different strategies normalized to that of DYN-PROG-COUNT$(X, 0.1)$ when $C = R$ vary from 60 to 3600 seconds, for all distributions in Table 11.1 with support considered in hours with HPC cost function. We indicate in brackets the mean μ of each distribution.



(a) $C = R = 6\text{min}$



(b) $C = R = 30\text{min}$



(c) $C = R = 60\text{min}$

Figure C.2: Expected cost of DYN-PROG-COUNT(X, ε) as a function of ε for different distributions for X with HPC cost function. $C = R$ are set to $6, 30$ and 60min.

by that of DYN-PROG-COUNT ($\varepsilon = 0.1$), when $C = R = 360$s (resp. $C = R = 1$h) using HPC cost function. For each distribution: the second columns shows the best period found when τ varies from 1 to 1000 (with its associated cost normalized by that of DYN-PROG-COUNT), and the other columns present results for specific values of τ in that interval. Overall, results show similar trends when using the HPC cost function in compared with Tables 11.5 and 11.6. Periodic heuristics are often not able to reach the performance of DYN-PROG-COUNT, that take the benefits of its close approximation to an optimal solution.

Distribution	ALL-CHECKPOINT-PERIODIC							NO-CHECKPOINT-PERIODIC						
	Best τ	$\tau=1$	$\tau=200$	$\tau=400$	$\tau=600$	$\tau=800$	$\tau=1000$	Best τ	$\tau=1$	$\tau=200$	$\tau=400$	$\tau=600$	$\tau=800$	$\tau=1000$
Exponential	21 (1.00)	6.38	2.43	4.20	5.97	7.74	9.52	10 (1.35)	6.38	8.14	15.59	23.04	30.50	37.95
Weibull	300 (1.04)	62.32	1.07	1.05	1.12	1.21	1.31	64 (2.29)	62.32	3.53	6.03	8.60	11.19	13.79
Gamma	12 (1.01)	4.03	3.83	6.98	10.13	13.28	16.44	6 (1.24)	4.03	10.80	20.89	30.99	41.08	51.18
Lognormal	4 (1.07)	2.42	3.74	6.63	9.52	12.41	15.30	3 (1.17)	2.42	16.55	32.24	47.93	63.62	79.32
Pareto	999 (1.00)	128.29	1.39	1.11	1.03	1.01	1.00	415 (1.28)	128.29	1.45	1.28	1.32	1.41	1.51
TruncatedNormal	2 (1.61)	1.61	6.93	12.34	17.75	23.16	28.57	1 (1.61)	1.61	207.16	413.80	620.45	827.09	1033.73
Uniform	6 (1.01)	1.28	2.98	5.07	7.17	9.26	11.36	1 (1.28)	1.28	54.39	107.91	161.42	214.93	268.44
Beta	1 (1.03)	1.03	32.95	65.29	97.62	129.96	162.29	1 (1.03)	1.03	45.95	91.22	136.49	181.76	227.03
BoundedPareto	26 (1.01)	4.69	1.74	2.68	3.62	4.57	5.52	10 (1.38)	4.69	7.22	13.62	20.03	26.44	32.85

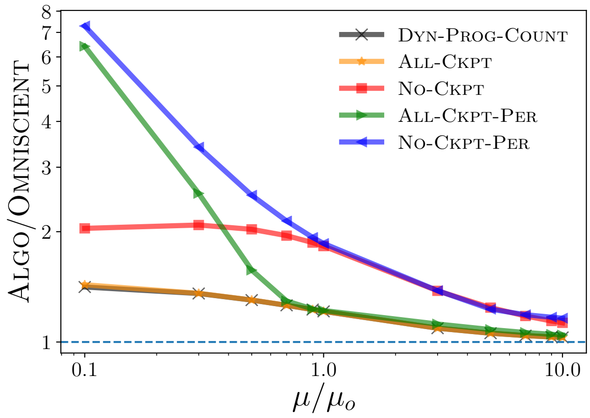
Table C.1: Performance of ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized by DYN-PROG-COUNT$(X, 0.1)$ for $C = R = 360$s, with HPC cost function.

Distribution	ALL-CHECKPOINT-PERIODIC							NO-CHECKPOINT-PERIODIC						
	Best τ	$\tau=1$	$\tau=200$	$\tau=400$	$\tau=600$	$\tau=800$	$\tau=1000$	Best τ	$\tau=1$	$\tau=200$	$\tau=400$	$\tau=600$	$\tau=800$	$\tau=1000$
Exponential	9 (1.25)	4.81	11.24	22.19	33.15	44.11	55.07	10 (1.02)	4.81	6.13	11.74	17.35	22.96	28.58
Weibull	116 (1.19)	44.78	1.30	1.83	2.42	3.03	3.65	64 (1.65)	44.78	2.54	4.33	6.18	8.04	9.91
Gamma	6 (1.29)	3.26	21.21	42.11	63.02	83.93	104.84	6 (1.00)	3.26	8.73	16.89	25.04	33.20	41.36
Lognormal	4 (1.13)	2.12	21.39	42.15	62.91	83.67	104.43	3 (1.03)	2.12	14.50	28.24	41.99	55.73	69.48
Pareto	438 (1.11)	103.29	1.27	1.11	1.13	1.21	1.31	415 (1.03)	103.29	1.17	1.03	1.06	1.13	1.21
TruncatedNormal	1 (1.45)	1.45	41.13	81.05	120.98	160.91	200.83	1 (1.45)	1.45	186.46	372.46	558.46	744.46	930.46
Uniform	2 (1.00)	1.08	15.21	29.71	44.21	58.71	73.22	1 (1.08)	1.08	45.90	91.05	136.21	181.36	226.52
Beta	1 (1.03)	1.03	263.43	526.83	790.22	1053.62	1317.02	1 (1.03)	1.03	45.65	90.62	135.59	180.56	225.53
BoundedPareto	11 (1.14)	3.64	6.50	12.51	18.53	24.55	30.57	10 (1.07)	3.64	5.59	10.56	15.53	20.50	25.46

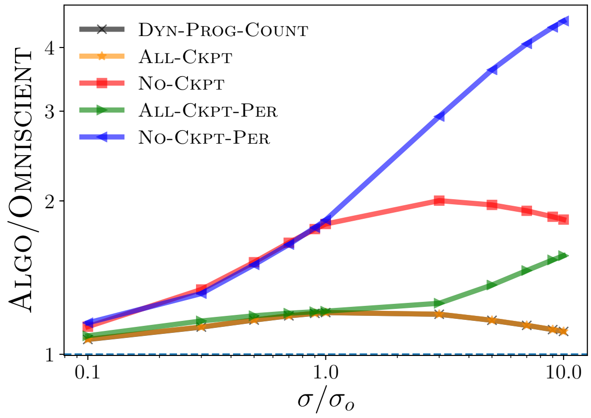
Table C.2: Performance of ALL-CHECKPOINT-PERIODIC and NO-CHECKPOINT-PERIODIC, normalized by DYN-PROG-COUNT$(X, 0.1)$ for $C = R = 1$h, with HPC cost function.

C.2 Results for Scenario 2

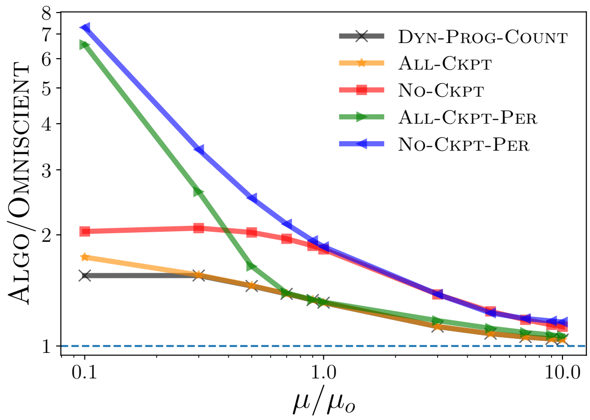
Figure C.3 shows similar trends using the same setup with the HPC cost function.



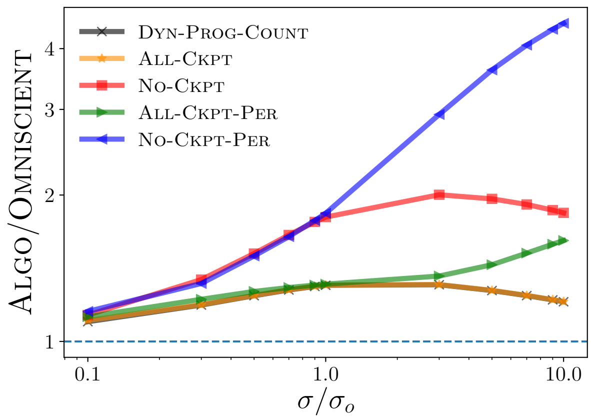
(a) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 600$s



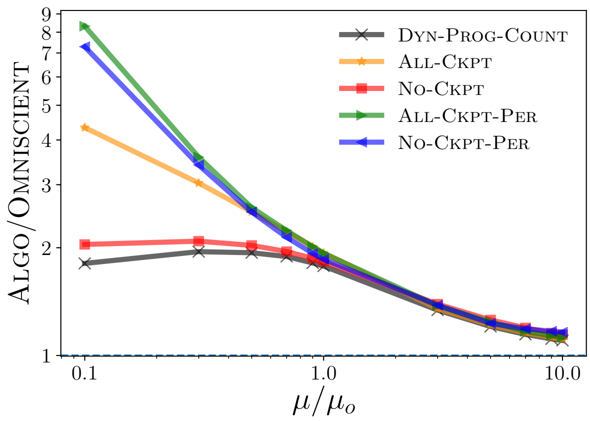
(b) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 600$s



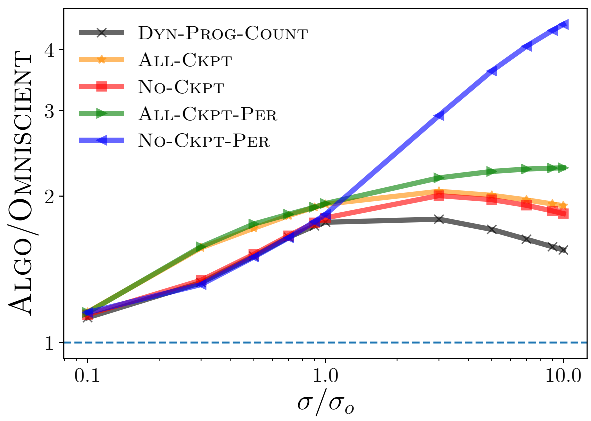
(c) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 1$h



(d) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 1$h



(e) Variation of μ, $\sigma = \sigma_o = 19.7$h, $C = R = 12$h



(f) Variation of σ, $\mu = \mu_o = 21.4$h, $C = R = 12$h

Figure C.3: Normalized performance of algorithms with omniscient scheduler when μ or σ vary, using HPC cost function ($\alpha = \beta = 1.0$, $\gamma = 0$). Basis is the LogNormal distribution in Figure 9.2 ($\mu_o = 21.4$h, $\sigma_0 = 19.7$h). $C = R$ are set to $600, 3600$ and 43200s (12h), $\varepsilon = 1$.

Appendix D

Description of the different platforms for stochastic application experiments

In this appendix, we present the different platforms used to perform the simulation process and experiments in the second part of the manuscript.

D.1 Haswell platform

We used an Haswell platform hosted on Plafrim[1], a development platform located in Inria Bordeaux-Sud Ouest. Each node of the platform is composed of two Intel Xeon E5-2680v3 processors with 12 cores each, with associated frequency of 2,5 GHz. Figure D.1 describes the topology of this platform generated by the *hwloc* library.

D.2 Knights Landing platform

To perform experiments with checkpoints in Chapter 12, we used a *KNL* platform composed of a 256-thread Intel Xeon Phi 7230 processor (Knights Landing), running at 1.30GHz frequency. It is configured Quadrant/Cache mode with 96GB of main memory. Figure D.2 presents an abstraction of the topology of this node, generated with the *hwloc* library.

[1] https://www.plafrim.fr/

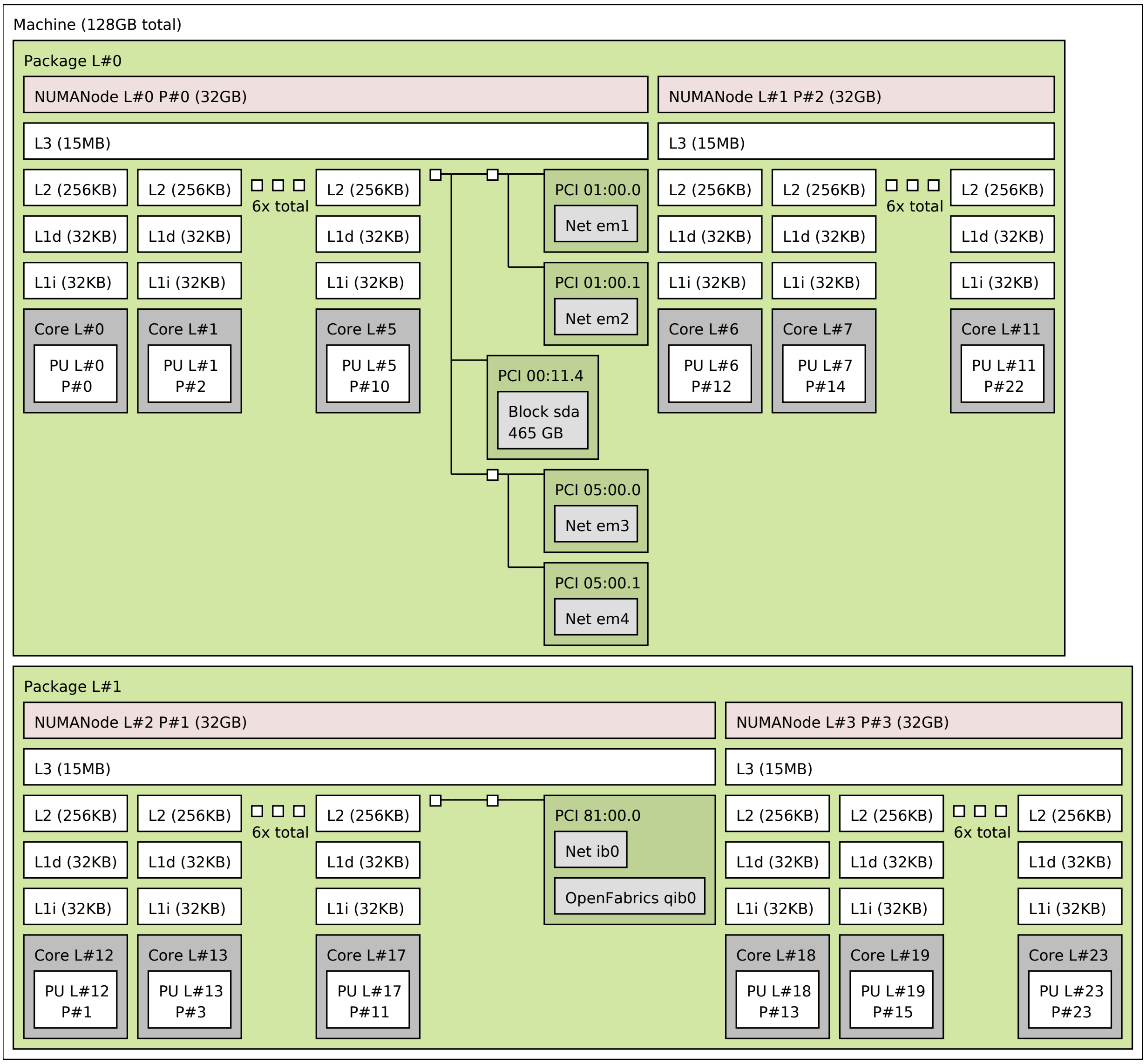


Figure D.1: Topology of the *Haswell* platform made of dual-processor Haswell Xeon E5-2680v3 nodes.

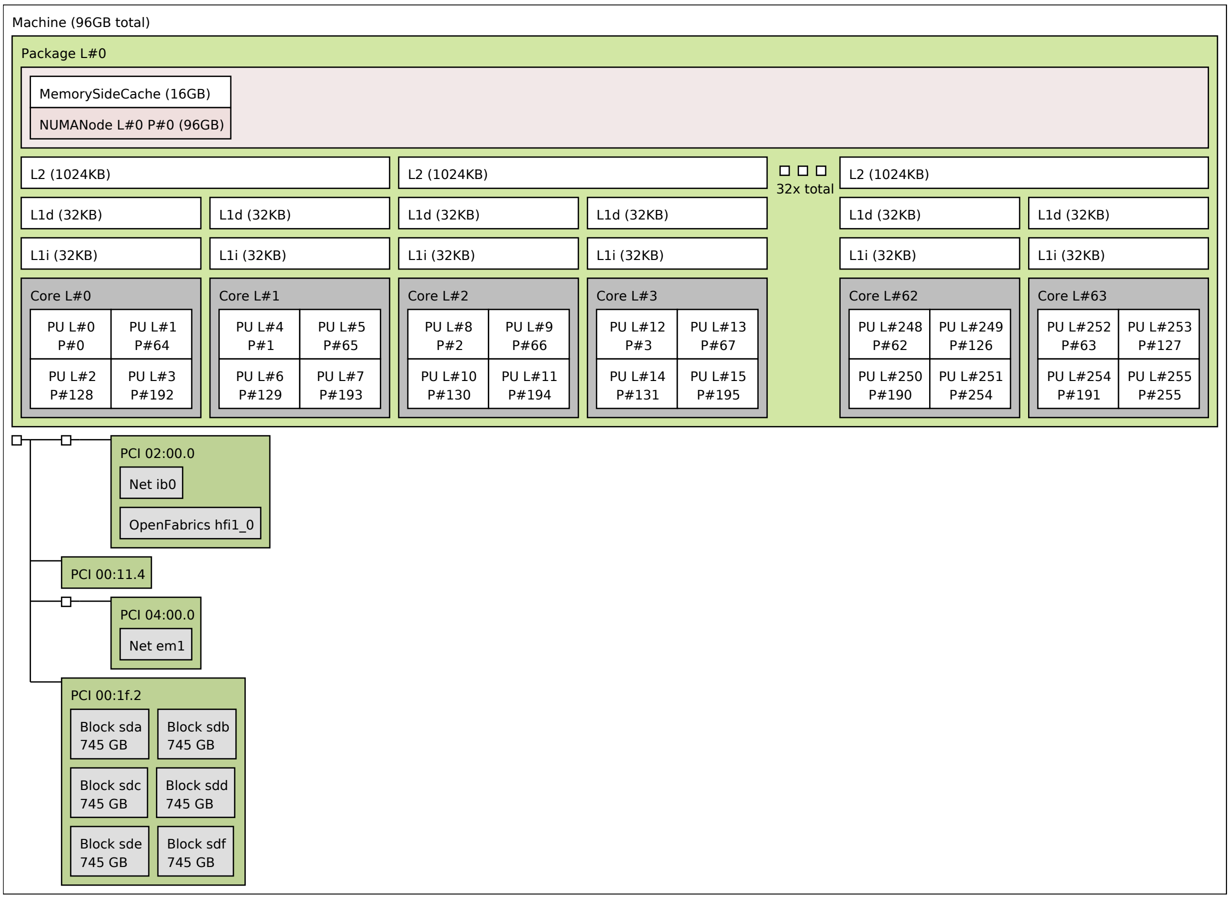


Figure D.2: Topology of the *KNL* platform made of a *Knights Landing* Xeon Phi 7230 processor.

Acronyms

AI Artificial Intelligence. 17

CPU Central Processing Unit. 14

FCN Fully Convolutional Network. 82

FLOPS FLoating-Point Operations Per Seconds. 11

FPTAS Fully Polynomial-Time Approximation Scheme. 105

GPU Graphical Processing Unit. 14

HACC Hardware/Hybrid Accelerated Cosmology Code. 17

HPC High Performance Computing. 11

hwloc Portable Hardware Locality package. 30

I/O Input/Output operation. 14

MPI Message Passing Interface. 17

MRI Magnetic Resonance Imaging. 81

NVRAM Non-volatile Random Access Memory. 28

PFS Parallel File System. 14

RAM Random Access Memory. 29

SLANT Spatially Localized Atlas Network Tiles. 81

SSD Solid-state Drive. 28

References

[1] Checkpoint Restart in Userspace (CRIU). https://criu.org.

[2] Google Protocol Buffer Format. https://developers.google.com/protocol-buffers.

[3] Gromacs versatile package to perform molecular dynamics. http://www.gromacs.org/. Accessed: 2020-07-01.

[4] List of the 500 most powerful computer systems in the world. https://www.top500.org/. Accessed: 2020-01-07.

[5] Presentation of Aurora exascale machine. https://www.energy.gov/articles/us-department-energy-and-intel-build-first-exascale-supercom Accessed: 2020-07-01.

[6] Specifications of Aurora exascale machine. https://aurora.alcf.anl.gov/. Accessed: 2020-07-01.

[7] Specifications of Frontier exascale machine. https://www.olcf.ornl.gov/frontier/#1. Accessed: 2020-07-01.

[8] Specifications of Jaguar supercomputer, ranked first at Top500 in June 2010. https://web.archive.org/web/20091207231123/http://www.nccs.gov/computing-resources/jaguar/#XT5-6-Core-Upgrade. Accessed: 2020-07-15.

[9] Specifications of Summit supercomputer, ranked first at Top500 in November 2019. https://www.olcf.ornl.gov/summit/. Accessed: 2020-07-15.

[10] U.S. Energy Information Administration. https://www.eia.gov/. Accessed: 2020-07-01.

[11] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan et Scott Klasky : Just in Time: Adding Value to the IO Pipelines of High Performance Applications with JITStaging. *In International symposium on High performance distributed computing*, pages 27–36, 2011.

[12] Hasan ABBASI, Matthew WOLF, Greg EISENHAUER, Scott KLASKY, Karsten SCHWAN et Fang ZHENG : DataStager: Scalable Data Staging Services for Petascale Applications. *In 18th ACM international symposium on High performance distributed computing*, pages 39–48, 2009.

[13] Bilge ACUN : *Mitigating Variability in HPC Systems and Applications for Performance and Power Efficiency*. Thèse de doctorat, University of Illinois at Urbana-Champaign, 2017.

[14] Maxim AFANASYEV et Haim MENDELSON : Service provider competition: Delay cost structure, segmentation, and cost advantage. *Manufacturing & Service Operations Management*, 12(2):213–235, 2010.

[15] Sean AHERN, Arie SHOSHANI, Kwan-Liu MA, Alok CHOUDHARY, Terence CRITCHLOW, Scott KLASKY, Valerio PASCUCCI, Jim AHRENS, E Wes BETHEL, Hank CHILDS *et al.* : Scientific discovery at the exascale. Report from the DOE ASCR 2011 Workshop on Exascale Data Management. *Analysis, and Visualization*, 2(3), 2011.

[16] AMAZON : AWS pricing information. https://aws.amazon.com/ec2/pricing/. Accessed: 2020-07-11.

[17] G. AMDAHL : The validity of the single processor approach to achieving large scale computing capabilities. *In AFIPS Conference Proceedings*, pages 483–485, 1967.

[18] Dan ANDRESEN, William HSU, Huichen YANG et Adedolapo OKANLAWON : Machine Learning for Predictive Analytics of Compute Cluster Jobs. *CoRR*, abs/1806.01116, 2018.

[19] Jason ANSEL, Kapil ARYA et Gene COOPERMAN : DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. *In 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–12, Rome, Italy, 2009. IEEE.

[20] Cédric AUGONNET, Samuel THIBAULT, Raymond NAMYST et Pierre-André WACRENIER : StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[21] Guillaume AUPY, Olivier BEAUMONT et Lionel EYRAUD-DUBOIS : What Size Should your Buffers to Disks be? *In International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, Canada, mai 2018. IEEE.

[22] Guillaume AUPY, Olivier BEAUMONT et Lionel EYRAUD-DUBOIS : Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention. *In IPDPS 2019 - 33rd IEEE International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brazil, mai 2019.

[23] Guillaume AUPY, Anne BENOIT, Thomas HÉRAULT, Yves ROBERT, Frédéric VIVIEN et Dounia ZAIDOUNI : On the Combination of Silent Error Detection and Checkpointing. *In IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013*, pages 11–20, 2013.

[24] Guillaume AUPY, Yves ROBERT, Frédéric VIVIEN et Dounia ZAIDOUNI : Checkpointing algorithms and fault prediction. *J. Parallel Distrib. Comput.*, 74(2):2048–2064, 2014.

[25] Utkarsh AYACHIT, Brad WHITLOCK, Matthew WOLF, Burlen LORING, Berk GEVECI, David LONIE et E. Wes BETHEL : The SENSEI Generic in Situ Interface. *In Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, ISAV '16, pages 40–44, Piscataway, NJ, USA, 2016. IEEE Press.

[26] Michael BAUER, Sean TREICHLER, Elliott SLAUGHTER et Alex AIKEN : Legion: Expressing Locality and Independence with Logical Regions. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[27] L. BAUTISTA-GOMEZ, S. TSUBOI, D. KOMATITSCH, F. CAPPELLO, N. MARUYAMA et S. MATSUOKA : FTI: High performance Fault Tolerance Interface for hybrid systems. *In SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[28] Pierre-Louis BAZIN, Jennifer L. CUZZOCREO, Michael A. YASSA, William GANDLER, Matthew J. MCAULIFFE, Susan S. BASSETT et Dzung L. PHAM : Volumetric neuroimage analysis extensions for the MIPAV software package. *Journal of Neuroscience Methods*, 165(1):111 – 121, 2007.

[29] Tal BEN-NUN et Torsten HOEFLER : Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR*, abs/1802.09941, 2018.

[30] Janine C BENNETT, Hasan ABBASI, Peer-Timo BREMER, Ray GROUT, Attila GYULASSY, Tong JIN, Scott KLASKY, Hemanth KOLLA, Manish PARASHAR, Valerio PASCUCCI *et al.* : Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49. IEEE Computer Society Press, 2012.

[31] John BIDDISCOMBE, Jerome SOUMAGNE, Guillaume OGER, David GUIBERT et Jean-Guillaume PICCINALI : Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File

Driver. *In* Torsten KUHLEN, Renato PAJAROLA et Kun ZHOU, éditeurs : *Eurographics Symposium on Parallel Graphics and Visualization*, pages 91–100, 2011. Honourable Mention Award.

[32] J. BREITBART, S. PICKARTZ, S. LANKES, J. WEIDENDORFER et A. MONTI : Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization. *In 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017.

[33] François BROQUEDIS, Jérôme CLET-ORTEGA, Stéphanie MOREAUD, Nathalie FURMENTO, Brice GOGLIN, Guillaume MERCIER, Samuel THIBAULT et Raymond NAMYST : hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. *In* IEEE, éditeur : *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, février 2010.

[34] J. BRUNO, P. DOWNEY et G. N. FREDERICKSON : Sequencing Tasks with Exponential Service Times to Minimize the Expected Flow Time or Makespan. *Journal of the ACM*, 28(1):100–113, 1981.

[35] Louis-Claude CANON, Aurélie Kong Win CHANG, Yves ROBERT et Frédéric VIVIEN : Scheduling independent stochastic tasks under deadline and budget constraints. Research Report 9178, INRIA, juin 2018.

[36] Louis-Claude CANON et Emmanuel JEANNOT : Evaluation and optimization of the robustness of dag schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010.

[37] Nicolas CAPIT, Georges DA COSTA, Yiannis GEORGIOU, Guillaume HUARD, Cyrille MARTIN, Grégory MOUNIÉ, Pierre NEYRON et Olivier RICHARD : A batch scheduler with high level components. *In Cluster computing and Grid 2005 (CCGrid05)*, Cardiff, United Kingdom, 2005. IEEE.

[38] Julien CAPUL, Sébastien MORAIS et Jacques-Bernard LEKIEN : PaDaWAn: A Python Infrastructure for Loosely Coupled in Situ Workflows. *In Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '18, pages 7–12, New York, NY, USA, 2018. ACM.

[39] Shi CHEN, Hau LEE et Kamran MOINZADEH : Pricing Schemes in Cloud Computing: Utilization-Based versus Reservation-Based. *Production and Operations Management*, 2017.

[40] Yang CHEN : Checkpoint and Restore of Micro-service in Docker Containers. *In 2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*, pages 915–918. Atlantis Press, 2015/10.

[41] J. T. DALY : "A higher order estimate of the optimum checkpoint interval for restart dumps". *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[42] Jeffrey DEAN et Sanjay GHEMAWAT : MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, janvier 2008.

[43] Ewa DEELMAN, Gurmeet SINGH, Mei-Hui SU, James BLYTHE, Yolanda GIL, Carl KESSELMAN, Gaurang MEHTA, Karan VAHI, G. Bruce BERRIMAN, John GOOD, Anastasia LAITY, Joseph C. JACOB et Daniel S. KATZ : Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Sci. Program.*, 13(3):219–237, juillet 2005.

[44] Ewa DEELMAN, Karan VAHI, Gideon JUVE, Mats RYNGE, Scott CALLAGHAN, Philip J MAECHLING, Rajiv MAYANI, Weiwei CHEN, Rafael Ferreira da SILVA, Miron LIVNY et Kent WENGER : Pegasus: a Workflow Management System for Science Automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[45] Ludwig DIERKS et Sven SEUKEN : Cloud pricing: the spot market strikes back. *In The Workshop on Economics of Cloud Computing*, 2016.

[46] Estelle DIRAND, Laurent COLOMBET et Bruno RAFFIN : TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics. *In SCA18 - Supercomputing Frontiers Asia 2018*, Singapore, Singapore, mars 2018.

[47] Ciprian DOCAN, Manish PARASHAR et Scott KLASKY : DART: a substrate for high speed asynchronous data IO. *In 17th international symposium on High performance distributed computing*, pages 219–220, 2008.

[48] Ciprian DOCAN, Manish PARASHAR et Scott KLASKY : DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. *Cluster Computing*, 15(2):163–181, 2012.

[49] F. DONG, J. LUO, A. SONG et J. JIN : Resource Load Based Stochastic DAGs Scheduling Mechanism for Grid Environment. *In IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 197–204, 2010.

[50] Matthieu DORIER, Gabriel ANTONIU, Franck CAPPELLO, Marc SNIR et Leigh ORF : Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O. *In CLUSTER - IEEE International Conference on Cluster Computing*, Beijing, China, septembre 2012. IEEE.

[51] Matthieu DORIER, Gabriel ANTONIU, Robert ROSS, Dries KIMPE et Shadi IBRAHIM : CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. *In IPDPS - International Parallel and Distributed Processing Symposium*, Phoenix, United States, mai 2014.

[52] Matthieu DORIER, Roberto SISNEROS, Tom PETERKA, Gabriel ANTONIU et Dave SEMERARO : Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework. *In IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2013.

[53] M. DREHER et B. RAFFIN : A Flexible Framework for Asynchronous in Situ and in Transit Analytics for Scientific Simulations. *In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 277–286, May 2014.

[54] Matthieu DREHER et Tom PETERKA : Bredala: Semantic Data Redistribution for In Situ Applications. *In Proceedings of IEEE Cluster 2016*. IEEE, 2016.

[55] Ifeanyi P EGWUTUOHA, David LEVY, Bran SELIC et Shiping CHEN : A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[56] N. FABIAN, K. MORELAND, D. THOMPSON, A. C. BAUER, P. MARION, B. GEVECIK, M. RASQUIN et K. E. JANSEN : The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library. *In 2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 89–96, Oct 2011.

[57] D FEITELSON : Workload modeling for computer systems performance evaluation. *Version 1.0.3*, pages 1–607, 2014.

[58] Lee FRIEDMAN et Gary H. GLOVER : Report on a multicenter fMRI quality assurance protocol. *Journal of Magnetic Resonance Imaging*, 23(6):827–839, 2006.

[59] Ana GAINARU et Guillaume PALLEZ : Making Speculative Scheduling Robust to Incomplete Data. *In ScalA19: 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Denver, United States, novembre 2019.

[60] Ana GAINARU, Guillaume PALLEZ, Hongyang SUN et Padma RAGHAVAN : Speculative Scheduling for Stochastic HPC Applications. *In ICPP 2019 - 48th International Conference on Parallel Processing*, Kyoto, Japan, août 2019.

[61] Ana GAINARU, Hongyang SUN, Guillaume AUPY, Yuankai HUO, Bennett A LANDMAN et Padma RAGHAVAN : On-the-fly scheduling vs. reservation-based scheduling for unpredictable workflows. *International Journal of High Performance Computing Applications*, 2019.

[62] Bruno GAUJAL et Jean-Marc VINCENT : Comparisons of Stochastic Task-Resource Systems. *In Introduction to Scheduling*, page Chapter 10. Springer, 2009.

[63] E. GAUSSIER, J. LELONG, V. REIS et D. TRYSTRAM : Online Tuning of EASY-Backfilling using Queue Reordering Policies. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2304–2316, 2018.

[64] Ashish GOEL et Piotr INDYK : Stochastic Load Balancing and Related Problems. *In FOCS*, pages 579–586. ACM, 1999.

[65] GOOGLE : GCP pricing information. https://cloud.google.com/pricing/. Accessed: 2020-07-11.

[66] Jian GUO, Akihiro NOMURA, Ryan BARTON, Haoyu ZHANG et Satoshi MATSUOKA : Machine Learning Predictions for Underestimation of Job Runtime on HPC System. *In* Rio YOKOTA et Weigang WU, éditeurs : *Supercomputing Frontiers*, pages 179–198, Cham, 2018. Springer International Publishing.

[67] A. GUPTA, P. FARABOSCHI, F. GIOACHIN, L. V. KALE, R. KAUFMANN, B. LEE, V. MARCH, D. MILOJICIC et C. H. SUEN : Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, July 2016.

[68] John L. GUSTAFSON : Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, mai 1988.

[69] Salman HABIB, Adrian POPE, Hal FINKEL, Nicholas FRONTIERE, Katrin HEITMANN, David DANIEL, Patricia FASEL, Vitali MOROZOV, George ZAGARIS, Tom PETERKA et et AL. : HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, Jan 2016.

[70] Paul H. HARGROVE et Jason C. DUELL : Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics. Conference Series*, 46, 9 2006.

[71] Robert L. HARRIGAN, Benjamin C. YVERNAULT, Brian D. BOYD, Stephen M. DAMON, Kyla David GIBNEY, Benjamin N. CONRAD, Nicholas S. PHILLIPS, Baxter P. ROGERS, Yurui GAO et Bennett A. LANDMAN : Vanderbilt University Institute of Imaging Science Center for Computational Imaging XNAT: A multimodal data archive and processing environment. *NeuroImage*, 124:1097–1101, 2016.

[72] Tim HARRIS, Martin MAAS et Virendra J. MARATHE : Callisto: Co-scheduling Parallel Runtime Systems. *In Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14*, pages 24:1–24:14, 2014.

[73] James HAXBY, Jyothi Swaroop GUNTUPALLI, Andrew CONNOLLY, Yaroslav HALCHENKO, Bryan CONROY, Maria GOBBINI, Michael HANKE et Peter RAMADGE : A Common, High-Dimensional Model of the Representational Space in Human Ventral Temporal Cortex. *Neuron*, 72:404–16, 10 2011.

[74] A. HEIRICH, E. SLAUGHTER, M. PAPADAKIS, W. LEE, T. BIEDERT et A. AIKEN : In Situ Visualization with Task-based Parallelism. *In Workshop on In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV'17)*, 2017.

[75] Alan HEIRICH, Elliott SLAUGHTER, Manolis PAPADAKIS, Wonchan LEE, Tim BIEDERT et Alex AIKEN : In situ visualization with task-based parallelism. *In Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 17–21. ACM, 2017.

[76] Maurice HERLIHY et Nir SHAVIT : *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[77] Benjamin HINDMAN, Andy KONWINSKI, Matei ZAHARIA, Ali GHODSI, Anthony D. JOSEPH, Randy KATZ, Scott SHENKER et Ion STOICA : Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. *In 8th USENIX Conf. Networked Systems Design and Implementation*, pages 295–308, 2011.

[78] Reazul HOQUE, Thomas HERAULT, George BOSILCA et Jack DONGARRA : Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. *In Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, pages 6:1–6:8, New York, NY, USA, 2017. ACM.

[79] Yuankai HUO, Zhoubing XU, Katherine ABOUD, Prasanna PARVATHANENI, Shunxing BAO, Camilo BERMUDEZ, Susan M. RESNICK, Laurie E. CUTTING et Bennett A. LANDMAN : Spatially Localized Atlas Network Tiles Enables 3D Whole Brain Segmentation from Limited Data". *In* Alejandro F. FRANGI, Julia A. SCHNABEL, Christos DAVATZIKOS, Carlos ALBEROLA-LÓPEZ et Gabor FICHTINGER, éditeurs : *Medical Image Computing and Computer Assisted Intervention – MICCAI 2018"*, pages 698–705, Cham, 2018. Springer International Publishing.

[80] Yuankai HUO, Zhoubing XU, Yunxi XIONG, Katherine ABOUD, Prasanna PARVATHANENI, Shunxing BAO, Camilo BERMUDEZ, Susan M. RESNICK, Laurie E. CUTTING et Bennett A. LANDMAN : 3D Whole Brain Segmentation using Spatially Localized Atlas Network Tiles. *NeuroImage*, 194:105 – 119, 2019.

[81] Zaeem HUSSAIN, Taieb ZNATI et Rami MELHEM : Partial Redundancy in HPC Systems with Non-Uniform Node Reliabilities. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[82] Thomas HÉRAULT et Yves ROBERT, éditeurs. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag, 2015.

[83] Yuichi INADOMI, Tapasya PATKI, Koji INOUE, Mutsumi AOYAGI, Barry ROUNTREE, Martin SCHULZ, David LOWENTHAL, Yasutaka WADA, Keiichiro FUKAZAWA, Masatsugu UEDA *et al.* : Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. *In SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[84] Michael ISARD, Mihai BUDIU, Yuan YU, Andrew BIRRELL et Dennis FETTERLY : Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. *In 2nd ACM SIGOPS/EuroSys European Conf. Computer Systems*, 2007.

[85] Leila ISMAIL et Latifur KHAN : Implementation and performance evaluation of a scheduling algorithm for divisible load parallel applications in a cloud computing environment. *Software: Practice and Experience*, 45, 2014.

[86] Hartmut KAISER, Thomas HELLER, Bryce ADELSTEIN-LELBACH, Adrian SERIO et Dietmar FEY : HPX: A Task Based Programming Model in a Global Address Space. *In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS'14)*, 2014.

[87] Jon KLEINBERG, Yuval RABANI et Éva TARDOS : Allocating Bandwidth for Bursty Connections. *In STOC*, pages 664–673, 1997.

[88] Brian KOCOLOSKI : *Scalability in the Presence of Variability*. Thèse de doctorat, University of Pittsburgh, 2018.

[89] Rajath KUMAR et Sathish VADHIYAR : Identifying Quick Starters: Towards an Integrated Framework for Efficient Predictions of Queue Waiting Times of Batch Parallel Jobs. *In* Walfredo CIRNE, Narayan DESAI, Eitan FRACHTENBERG et Uwe SCHWIEGELSHOHN, éditeurs : *Job Scheduling Strategies for Parallel Processing*, pages 196–215, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[90] Pamela J LAMONTAGNE, Tammie L.S. BENZINGER, John C. MORRIS, Sarah KEEFE, Russ HORNBECK, Chengjie XIONG, Elizabeth GRANT, Jason HASSENSTAB, Krista MOULDER, Andrei VLASSENKO, Marcus E. RAICHLE, Carlos CRUCHAGA et Daniel MARCUS : OASIS-3: Longitudinal Neuroimaging, Clinical, and Cognitive Dataset for Normal Aging and Alzheimer Disease. *medRxiv*, 2019.

[91] Bennett LANDMAN : Medical-image Analysis and Statistical Interpretation (MASI) Lab. `https://my.vanderbilt.edu/masi/`.

[92] Samuel LANG, Philip CARNS, Robert LATHAM, Robert ROSS, Kevin HARMS et William ALLCOCK : I/O Performance Challenges at Leadership Scale. *In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[93] Matthew LARSEN, James AHRENS, Utkarsh AYACHIT, Eric BRUGGER, Hank CHILDS, Berk GEVECI et Cyrus HARRISON : The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. *In Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 42–46. ACM, 2017.

[94] Matthew LARSEN, Cyrus HARRISON, James KRESS, David PUGMIRE, Jeremy S. MEREDITH et Hank CHILDS : Performance Modeling of In Situ Rendering. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, pages 24:1–24:12, Salt Lake City, UT, novembre 2016.

[95] K. LI, X. TANG, B. VEERAVALLI et K. LI : Scheduling Precedence Constrained Stochastic Tasks on Heterogeneous Cluster Systems. *IEEE Transactions on Computers*, 64(1):191–204, 2015.

[96] Min LI, Sudharshan S. VAZHKUDAI, Ali R. BUTT, Fei MENG, Xiaosong MA, Youngjae KIM, Christian ENGELMANN et Galen SHIPMAN : Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures. *In International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2010.

[97] Shigang LI, Tal BEN-NUN, Salvatore Di GIROLAMO, Dan ALISTARH et Torsten HOEFLER : Taming unbalanced training workloads in deep learning with partial collective operations. *In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–61, 2020.

[98] David A. LIFKA : The ANL/IBM SP Scheduling System. *In JSSPP*, pages 295–303, 1995.

[99] Ning LIU, Jason COPE, Philip CARNS, Christopher CAROTHERS, Robert ROSS, Gary GRIDER, Adam CRUME et Carlos MALTZAHN : On the role of burst buffers in leadership-class storage systems. *In 012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, apr 2012.

[100] Jay LOFSTEAD, Fang ZHENG, Qing LIU, Scott KLASKY, Ron OLDFIELD, Todd KORDENBROCK, Karsten SCHWAN et Matthew WOLF : Managing variability in the io performance of petascale storage systems. *In SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2010.

[101] Jay F. LOFSTEAD, Scott KLASKY, Karsten SCHWAN, Norbert PODHORSZKI et Chen JIN : Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). *In Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[102] Bertram LUDÄSCHER, Ilkay ALTINTAS, Chad BERKLEY, Dan HIGGINS, Efrat JAEGER, Matthew JONES, Edward A. LEE, Jing TAO et Yang ZHAO : Scientific Workflow Management and the Kepler System: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, août 2006.

[103] Kwan-Liu MA, Chaoli WANG, Hongfeng YU et Anna TIKHONOVA : In-situ processing and visualization for ultrascale simulations. *Journal of Physics: Conference Series*, 78(1):012043, 2007.

[104] Xiaosong MA, J. LEE et M. WINSLETT : High-Level Buffering for Hiding Periodic Output Cost in Scientific Simulations. *Parallel and Distributed Systems, IEEE Transactions on*, 17(3):193–204, 2006.

[105] Preeti MALAKAR, Venkatram VISHWANATH, Christopher KNIGHT, Todd MUNSON et Michael E. PAPKA : Optimal Execution of Co-analysis for Large-scale Molecular Dynamics Simulations. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 60:1–60:14, Piscataway, NJ, USA, 2016. IEEE Press.

[106] Preeti MALAKAR, Venkatram VISHWANATH, Todd MUNSON, Christopher KNIGHT, Mark HERELD, Sven LEYFFER et Michael E. PAPKA : Optimal Scheduling of In-situ Analysis for Large-scale Scientific Simulations. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 52:1–52:11, New York, NY, USA, 2015. ACM.

[107] A. MATSUNAGA et J. A. B. FORTES : On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. *In 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504, 2010.

[108] Andrea MECHELLI, Cathy J. PRICE, Karl J. FRISTON et John ASHBURNER : Voxel-based morphometry of the human brain: methods and applications. *Current Medical Imaging Reviews*, 1:105–113, 2005.

[109] André MERZKY, Mark SANTCROOS, Matteo TURILLI et Shantenu JHA : RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers. *CoRR*, abs/1512.08194, 2015.

[110] Andrey MIRKIN, Alexey KUZNETSOV et Kir KOLYSHKIN : Containers checkpointing and live migration. *In In Ottawa Linux Symposium*, 2008.

[111] Rolf H. MÖHRING, Andreas S. SCHULZ et Marc UETZ : Approximation in Stochastic Scheduling: The Power of LP-based Priority Policies. *Journal of the ACM*, 46(6):924–942, 1999.

[112] Clement MOMMESSIN, Matthieu DREHER, Bruno RAFFIN et Tom PETERKA : Automatic Data Filtering for In Situ Workflows . *In IEEE Cluster*, Hawaii, 2017.

[113] A. W. MU'ALEM et D. G. FEITELSON : Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[114] Ahuva W. MU'ALEM et Dror G. FEITELSON : Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[115] José Niño MORA : Stochastic Scheduling. *Encyclopedia of Optimization*, pages 3818–3824, 2009.

[116] Jun Woo PARK, Alexey TUMANOV, Angela JIANG, Michael A. KOZUCH et Gregory R. GANGER : 3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty. *In Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[117] Tapasya PATKI, David K LOWENTHAL, Barry ROUNTREE, Martin SCHULZ et Bronis R DE SUPINSKI : Exploring hardware overprovisioning in power-constrained, high performance computing. *In Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 173–182, 2013.

[118] Tapasya PATKI, Jayaraman J. THIAGARAJAN, Alexis AYALA et Tanzima Z. ISLAM : Performance Optimality or Reproducibility: That is the Question. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[119] P. PÉBAŸ, J. C. BENNETT, D. HOLLMAN, S. TREICHLER, P. S. MCCORMICK, C. M. SWEENEY, H. KOLLA et A. AIKEN : Towards Asynchronous Many-Task in Situ Data Analysis Using Legion. *In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1033–1037, May 2016.

[120] Simon PICKARTZ, Niklas EILING, Stefan LANKES, Lukas RAZIK et Antonello MONTI : "Migrating LinuX Containers Using CRIU". *In* Michela TAUFER, Bernd MOHR et Julian M. KUNKEL, éditeurs : *"High Performance Computing"*, pages 674–684, Cham, 2016. Springer International Publishing.

[121] Michael L. PINEDO : *Scheduling: Theory, Algorithms, and Systems*. Springer-Verlag New York, Inc., Third édition, 2008.

[122] Michael L. PINEDO : *Scheduling: Theory, Algorithms, and Systems*. Springer, 3rd édition, 2008.

[123] Xing PU, Ling LIU, Yiduo MEI, Sankaran SIVATHANU, Younggyun KOH et Calton PU : Understanding performance interference of i/o workload in virtualized cloud environments. *In 2010 IEEE 3rd International Conference on Cloud Computing*, pages 51–58. IEEE, 2010.

[124] Gonzalo Pedro RODRIGO ÁLVAREZ, Per-Olov ÖSTBERG, Erik ELMROTH, Katie ANTYPAS, Richard GERBER et Lavanya RAMAKRISHNAN : HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. *In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 57–60, New York, NY, USA, 2015. ACM.

[125] Manuel RODRÍGUEZ, J.A. MORÍÑIGO et R. MAYO-GARCÍA : When you have a hammer, everything looks like a nail - Checkpoint/restart in Slurm. SLURM User Group 2017.

[126] Luís Moura SILVA et João Gabriel SILVA : System-level versus user-defined checkpointing. *In Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pages 68–74. IEEE, 1998.

[127] Ajeet SINGH, Pavan BALAJI et Wu-chun FENG : GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading. *In International Conference on Parallel Processing*, pages 261–268, 2009.

[128] David SKINNER et William KRAMER : Understanding the causes of performance variability in hpc workloads. *In IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149. IEEE, 2005.

[129] Joseph SKOVIRA, Waiman CHAN, Honbo ZHOU et David A. LIFKA : The EASY - LoadLeveler API Project. *In JSSPP*, pages 41–47, 1996.

[130] Aishwarya SONI et Muzammil HASAN : Pricing schemes in cloud computing: A review. *International Journal of Advanced Computer Research*, 7:60–70, 02 2017.

[131] Garrick STAPLES : TORQUE Resource Manager. *In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 8–es, New York, NY, USA, 2006. Association for Computing Machinery.

[132] Pradeep SUBEDI, Philip DAVIS, Shaohua DUAN, Scott KLASKY, Hemanth KOLLA et Manish PARASHAR : Stacker: an autonomic data movement engine for extreme-scale data staging-based in situ workflows. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, page 73. IEEE Press, 2018.

[133] Qian SUN, Tong JIN, Melissa ROMANUS, Hoang BUI, Fan ZHANG, Hongfeng YU, Hemanth KOLLA, Scott KLASKY, Jacqueline CHEN et Manish PARASHAR : Adaptive Data Placement for Staging-based Coupled Scientific Workflows. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 65:1–65:12, New York, NY, USA, 2015. ACM.

[134] Mohammed TANASH, Brandon DUNN, Daniel ANDRESEN, William HSU, Huichen YANG et Adedolapo OKANLAWON : Improving HPC System Performance by Predicting Job Resources via Supervised Machine Learning. *In Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[135] K. TANG, P. HUANG, X. HE, T. LU, S. S. VAZHKUDAI et D. TIWARI : Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior. *In 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 87–98, 2017.

[136] W. TANG, Z. LAN, N. DESAI, D. BUETTNER et Y. YU : Reducing Fragmentation on Torus-Connected Supercomputers. *In 2011 IEEE International Parallel Distributed Processing Symposium*, pages 828–839. IEEE, 2011.

[137] Xiaoyong TANG, Kenli LI, Guiping LIAO, Kui FANG et Fan WU : A Stochastic Scheduling Algorithm for Precedence Constrained Tasks on Grid. *Future Gener. Comput. Syst.*, 27(8):1083–1091, 2011.

[138] Dingwen TAO, Sheng DI, Zizhong CHEN et Franck CAPPELLO : Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization. *In 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 1129–1139, 2017.

[139] Théophile TERRAZ, Alejandro RIBES, Yvan FOURNIER, Bertrand IOOSS et Bruno RAFFIN : Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files. *In The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, pages 1 – 14, Denver, United States, novembre 2017.

[140] H. TOPCUOGLU, S. HARIRI et Min-You WU : Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS*, 13(3):260–274, March 2002.

[141] Tiankai TU, Charles A. RENDLEMAN, David W. BORHANI, Ron O. DROR, Justin GULLINGSRUD, Morten Ø. JENSEN, John L. KLEPEIS, Paul MARAGAKIS, Patrick MILLER, Kate A. STAFFORD et David E. SHAW : A Scalable Parallel Framework for

Analyzing Terascale Molecular Dynamics Simulation Trajectories. *In Conference on Supercomputing*, pages 56:1–56:12, 2008.

[142] Courtenay T VAUGHAN et Simon David HAMMOND : Evaluating Production Load Balancing Functions for Adaptive Mesh Schemes using Mini-Applications. Rapport technique, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[143] Vinod Kumar VAVILAPALLI, Arun C. MURTHY, Chris DOUGLAS, Sharad AGARWAL, Mahadev KONAR, Robert EVANS, Thomas GRAVES, Jason LOWE, Hitesh SHAH, Siddharth SETH, Bikas SAHA, Carlo CURINO, Owen O'MALLEY, Sanjay RADIA, Benjamin REED et Eric BALDESCHWIELER : Apache Hadoop YARN: Yet Another Resource Negotiator. *In the 4th Annual Symposium on Cloud Computing*, pages 5:1–5:16, 2013.

[144] V. VISHWANATH, M. HERELD et M.E. PAPKA : Toward simulation-time data analysis and I/O acceleration on leadership-class systems. *In Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 9–14, Oct 2011.

[145] John von NEUMANN : First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, octobre 1993.

[146] Yi WANG, Gagan AGRAWAL, Tekin BICER et Wei JIANG : Smart: A MapReduce-like Framework for In-situ Scientific Analytics. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 51:1–51:12, New York, NY, USA, 2015. ACM.

[147] Gideon WEISS : Turnpike Optimality of Smith's Rule in Parallel Machines Stochastic Scheduling. *Math. Oper. Res.*, 17(2):255–270, 1992.

[148] Brad WHITLOCK, Jean M. FAVRE et Jeremy S. MEREDITH : Parallel in Situ Coupling of Simulation with a Fully Featured Visualization System. *In Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[149] Samuel WILLIAMS, Andrew WATERMAN et David PATTERSON : Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[150] K. WOLTER, éditeur. *Stochastic Models for Fault Tolerance, Restart, Rejuvenation, and Checkpointing*. Springer Verlag, 2010.

[151] H. XU et B. LI : Dynamic Cloud Pricing for Revenue Maximization. *IEEE Transactions on Cloud Computing*, 1(2):158–171, July 2013.

[152] Keiji YAMAMOTO et AL. : The K computer Operations: Experiences and Statistics. *Procedia Computer Science*, 29:576 – 585, 2014.

[153] O. YILDIZ, M. DORIER, S. IBRAHIM, R. ROSS et G. ANTONIU : On the root causes of cross-application i/o interference in hpc storage systems. *In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 750–759, 2016.

[154] Orcun YILDIZ, Amelie Chi ZHOU et Shadi IBRAHIM : Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC systems. *In Cluster'17-2017 IEEE International Conference on Cluster Computing*, Hawaii, United States, septembre 2017.

[155] Andy B. YOO, Morris A. JETTE et Mark GRONDONA : SLURM: Simple Linux Utility for Resource Management. *In JSSPP*, pages 44–60, 2003.

[156] Andy B YOO, Morris A JETTE et Mark GRONDONA : Slurm: Simple linux utility for resource management. *In Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[157] John W. YOUNG : A first order approximation to the optimum checkpoint interval. *Comm. ACM*, 17(9):530–531, 1974.

[158] Hongfeng YU, Chaoli WANG, R.W. GROUT, J.H. CHEN et Kwan-Liu MA : In situ visualization for large-scale combustion simulations. *Computer Graphics and Applications, IEEE*, 30(3):45–57, 2010.

[159] Fan ZHANG, C. DOCAN, M. PARASHAR, S. KLASKY, Norbert PODHORSZKI et Hasan ABBASI : Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform. *In Parallel Distributed Processing Symposium (IPDPS)*, pages 1352–1363, 2012.

[160] Zhaoning ZHANG, Lujia YIN, Yuxing PENG et Dongsheng LI : A quick survey on large scale distributed deep learning systems. *In 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1052–1056. IEEE, 2018.

[161] F. ZHENG, H. YU, C. HANTAS, M. WOLF, G. EISENHAUER, K. SCHWAN, H. ABBASI et S. KLASKY : Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. *In SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.

[162] F. ZHENG, H. ZOU, G. EISENHAUER, K. SCHWAN, M. WOLF, J. DAYAL, T. NGUYEN, J. CAO, H. ABBASI, S. KLASKY, N. PODHORSZKI et H. YU : FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. *In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 320–331, May 2013.

[163] Fang ZHENG, H. ABBASI, C. DOCAN, J. LOFSTEAD, Qing LIU, S. KLASKY, M. PARASHAR, N. PODHORSZKI, K. SCHWAN et M. WOLF : PreDatA - Preparatory Data Analytics on Peta-Scale Machines. *In Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[164] Sergey ZHURAVLEV, Sergey BLAGODUROV et Alexandra FEDOROVA : Addressing Shared Resource Contention in Multicore Processors via Scheduling. *In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 129–142, New York, NY, USA, 2010. Association for Computing Machinery.

[165] Salah ZRIGUI, Raphael de CAMARGO, Denis TRYSTRAM et Arnaud LEGRAND : Improving the performance of batch schedulers using online job size classification. 2019.

Publications

Articles in Peer-reviewed Conferences

[166] Guillaume AUPY, Ana GAINARU, Valentin HONORÉ, Padma RAGHAVAN, Yves ROBERT et Hongyang SUN : Reservation Strategies for Stochastic Jobs. *In IPDPS 2019 - 33rd IEEE International Parallel and Distributed Processing Symposium*, pages 166–175, Rio de Janeiro, Brazil, mai 2019. IEEE.

[167] Ana GAINARU, Brice GOGLIN, Valentin HONORÉ, Guillaume PALLEZ, Padma RAGHAVAN, Yves ROBERT et Hongyang SUN : Reservation and Checkpointing Strategies for Stochastic Jobs. *In IPDPS 2020 - 34th IEEE International Parallel and Distributed Processing Symposium*, New Orleans, United States, mai 2020.

[168] Valentin HONORÉ : Techniques d'ordonnancement pour les applications stochastiques sur plateformes HPC. *In COMPAS 2020*, Lyon, France, juin 2020.

Articles in Peer-reviewed Journals

[169] Guillaume AUPY, Brice GOGLIN, Valentin HONORÉ et Bruno RAFFIN : Modeling High-throughput Applications for in situ Analytics. *International Journal of High Performance Computing Applications*, 33(6):1185–1200, avril 2019.

[170] Ana GAINARU, Brice GOGLIN, Valentin HONORÉ et Guillaume PALLEZ : Profiles of upcoming HPC Applications and their Impact on Reservation Strategies. *IEEE Transactions on Parallel and Distributed Systems* , **To appear**.

Posters

[171] Valentin HONORÉ : Modeling HPC applications for in situ Analytics. IPDPS 2019 - 33rd IEEE International Parallel and Distributed Processing Symposium, mai 2019. Poster.

Research Reports

[172] Guillaume AUPY, Ana GAINARU, Valentin HONORÉ, Padma RAGHAVAN, Yves ROBERT et Hongyang SUN : Reservation Strategies for Stochastic Jobs (Extended Version). Research Report RR-9211, Inria & Labri, Univ. Bordeaux ; Department of EECS, Vanderbilt University, Nashville, TN, USA ; Laboratoire LIP, ENS Lyon & University of Tennessee Knoxville, Lyon, France, octobre 2018.

[173] Ana GAINARU, Brice GOGLIN, Valentin HONORÉ, Guillaume PALLEZ, Padma RAGHAVAN, Yves ROBERT et Hongyang SUN : Reservation and Checkpointing Strategies for Stochastic Jobs (Extended Version). Research Report RR-9294, Inria & Labri, Univ. Bordeaux ; Department of EECS, Vanderbilt University, Nashville, TN, USA ; Laboratoire LIP, ENS Lyon & University of Tennessee Knoxville, Lyon, France, octobre 2019.