Data-Aware Scheduling at Higher scale

Nicolas Vidal

LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE LaBRI

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Nicolas Vidal**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Dash - Data-Aware Scheduling at Higher Scale

Date de soutenance : Janvier 2022

Devant la commission d'examen composée de :

M. Emmanuel JEANNOT	Directeur de recherche, Inria / LaBRI	Directeur
M. Jay LOFSTEAD	Principal Member of Technical Staff, Sandia National Laboratories	Rapporteur
Mme. Alix MUNIER	Professeur, Sorbonne Université / LIP6	Rapporteuse
Mme. Sadaf ALAM	Chief Technology officer, CSCS / ETH Zurich	Examinatrice
M. Yves DENNELIN	Professeur, Inria / Grenoble INP	Examinateur
M. Guillaume PALLEZ	Chargé de recherche, Inria / LaBRI	Examinateur

– 2022 –

Résumé

Mots-clés Calcul haute performance, applications à forte intensité en données, 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 Dash - Data-Aware scheduling at higher scale

Abstract

Keywords High Performance Computing, scheduling algorithms, resource partitioning

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

Cette thèse a été menée au sein de l'équipe TADaaM au centre de recherche Inria de Bordeaux. Je tiens à remercier tout les collègues qui m'ont entourés pendant ces presques quatre ans. Tout d'abord Emmanuel Jeannot et Guillaume Pallez qui m'ont encadré. Leur présence ne s'est pas limité à leurs apports pourtant nombreux sur le plan scientifique mais aussi pas un lien humain conservé malgré les circonstances difficiles dues la pandémie. Dans l'équipe, mes pensées vont particulièrement à Valentin Honoré, ami cher et "grand frère de thèse", présence constante et réconfortante qui m'a conseillé et encouragé. Après son départ, j'ai pu compter sur Philippe Swartvagher, toujours au fait des différents impératifs et prêt à s'investir dans la vie de l'équipe et du centre. Guillaume Blin et Olivier Beaumont, membres de mon comité de suivi ont sû m'écouter lorsque le besoin (et les impératifs de l'école doctorales) s'imposait. Sur une échelle de temps plus longue, j'aimerai remercier Yves Robert pour ses enseignements et ses conseils depuis la licence. Enfin pour finir les remerciements professionnels, merci à tout les chercheurs et chercheuses, collègues, ami.e.s avec qui j'ai pu parler, partager un café, faire part de mon enthousiasme et de mes doutes au depuis 2018.

Mes remerciements vont au delà de l'équipe de travail, je pense également à mes proches et mes amis. Les membres de ma famille m'ont toujours encouragés et soutenus, j'en suis très touché et fier. Je pense à vous avec émotion, Christian, Régine, Violaine et en particulier mon frère Aurélien qui n'a pas pû être présent pour ma soutenance, accaparé par une nièce dont l'arrivée éclipse tout autre évènement de ce début d'année. J'ai hâte de te rencontrer Marta, tu as déjà une place bien à toi dans mes pensées. Enfin, j'ai conservé un lien très fort avec un groupe d'ami.e.s qui m'ont soutenus. Arthur Blot et Victor Lutfalla lors de rendez-vous à distance réguliers, Axel Kerinec à travers une correspondance parfois décousue mais toujours sincère. Alam, Amélie, Maria, Titouan et tout les autres, vous comptez vraiment à mes yeux. Pour finir, depuis plus d'un an, Léa Dislaire s'est fait une place dans ma vie, iel m'a encouragé, rassuré, de jour comme nuit. Merci d'être avec moi.

J'ai envie de partager la fierté et le soulagement d'avoir mené ce grand projet à bien avec chacun et chacune d'entre vous.

Résumé étendu en français

Introduction

Le calcul haute performance est le domaine scientifique entourant les plateformes de calcul à grande échelle que représentent les supercalculateurs. De nos jours, les scientifiques dépendent de ces machines afin d'exécuter des programmes qui ne peuvent pas être exécutés sur un simple ordinateur. Ces programmes atteignent des dimensions vertigineuses non seulement par la puissance de calcul nécessaire mais aussi par les quantités de données entrant en jeu. Les coûts de constructions et d'entretien de tels machines est justifié par leur utilisation efficace dans des domaines divers. Ainsi ces plateformes doivent être capable de gérer les différents besoins venant d'utilisateurs indépendants qui utilisent des sous ensemble de la machine indépendamment les uns les autres. Ces dernières années, les performances en terme de bande passante pour la gestion des données n'ont pas progressé au même rythme que l'augmentation de la puissance de calcul. En parallèle, le développement de l'analyse de données à grande échelle et de l'apprentissage en profondeur ont contribué à déplacer la focale d'une orientation calcul à une approche consciente des données. Malgré des changement architecturaux, l'utilisation de la bande passante n'est pas optimal dû au manque d'information sur l'état des ressources du calculateur et des accès à la bande passante des différentes applications en cours d'exécution. Par conséquent, des déplacements de données non cordonnés font perdre des opportunités d'optimiser globalement les entrées/sortes (E/S) conduisant à des opérations redondantes, de la contention et une perte performance. S'il n'est pas clair de savoir si la perte dû à la contention est pire que linéaire, des accès concurrents à la bande passante conduisent à des délais pour les applications devant attendre ou n'ayant qu'un accès partiel au système de fichier.

Cette thèse s'intéresse au problème global de la gestion des ressources afin d'éviter ce phénomène de goulot d'étranglement sur les données. D'autres travaux se sont déjà attaqué à cette problématique. Les approches logicielles proposées incluent la transformation des données en amont de leur transfert, l'inclusion de middleware servant d'intermédiaire entre le système de fichier et les applications afin de fluidifier les échanges. Du point de vue matériel, les constructeurs de machines peuvent inclure des mémoires tampons au niveau des nœuds de calcul afin de lisser les comportements par secousses des applications. D'autres solutions proposent de partitionner ou d'étager les architectures.

Nous avons choisi de traiter ce problème sur le plan théorique en le considérant comme un cas particulier d'ordonnancement à deux ressources. Tout au long de cette thèse, nous chercherons à définir comment modéliser des applications dans le cadre du calcul haute performance en se basant sur des résultats de expérimentaux peu adaptés à l'exercice théorique. Nous proposons des algorithmes afin de placer et ordonner les exécutions de ses application. Nous définirons aussi et surtout un protocole expérimental cherchant à évaluer de manière convaincante des modèles théoriques. Dans une première partie, nous nous intéressons à comment ordonner différentes tâches exclu-

sives sur la bande passante afin de fluidifier les opérations d'E/S. Dans une seconde partie, nous proposerons un algorithme de placement d'application prenant en compte leur besoins en terme de données. Enfin, nous comparerons les différentes manières d'exercer un contrôle de la bande passante en essayant de faire le lien entre différence qualitatives d'usage du réseau et caractéristiques du jeu d'application. Pour conclure, nous ouvrirons de nouvelles perspectives sur des stratégies hybrides combinant accès exclusifs et partage de bande passante.

Ordonnancement des accès à la bande passante

Dans ce premier chapitre, nous cherchons comment pouvons nous modéliser formellement le comportement des applications de calcul haute performance dans le cadre du problème d'engorgement des E/S puis comment ordonner les accès au système de fichier.

La première et fondamentale contribution de cette partie est la définition de modèles formels de plateforme et d'applications qui servent non seulement à définir et étudier le problème d'ordonnancement des opérations d'E/S mais qui sera adapté aux problèmes étendus des parties suivantes. Lorsque nous élargirons notre problématique au placement des applications sur les machines.

Modèle de plateforme Nous considérons une plateforme constituée de deux types de machines $\mathcal{A}$ (nœuds de calculs) et $\mathcal{B}$ (nœuds d'E/S). Dans le cadre de notre problème d'ordonnancement d'opérations d'E/S, nous supposons que les applications ont déjà été placé sur la machine en amont et qu'il n'y a pas de compétition à ce niveau. Ainsi, nous pouvons considérer que la machine $\mathcal{A}$ possède une quantité illimité de ressources de calcul. A contrario, la bande passante du système de fichier (machine $\mathcal{B}$) est partagée par toutes les applications. Il y en a donc un quantité limitée B, en normalisant, B=1. Nous appelons une telle instance *une plateforme bicolore*.

Modèle d'application Nous voulons prendre une compte un jeu d'applications scientifiques s'exécutant simultanément sur une plateforme parallèle. Les ressources de calculs ont déjà été allouées à chaque applications. En prenant en compte les E/S, celles-ci consistent en une alternance de phases *distinctes*: (i) un phase de calcul (sur la machine $\mathcal{A}$): (ii) une phase d'E/S (sur la machine $\mathcal{B}$). Formellement, un travail J_i est constitué de $2n_i$ opérations successives $A_{i,j}$,$B_{i,j}$ ($j \leqslant n_i$). La tâche $A_{i,j+1}$ (resp. $B_{i,j}$) peut seulement commencer quand l'opération $B_{i,j}$ (resp. $A_{i,j}$).

Nous pouvons ainsi définir le problème d'ordonnancement suivant: Étant donné un ensemble de travaux $J_i = (\prod_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$, un ordonnancement $\mathcal{S}$ est défini comme une permutations des tâches $((B_{i,j})_{j\leq n_i})_i$ qui satisfait, pour tous i, j,$B_{i,j}$ est avant $B_{i,j+1}$. En définissant C_i la fin du temps d'exécution d'une application J_i suivant l'ordonnancement $\mathcal{S}$ et le makespan $C^{\mathcal{S}}_{max}$ leur maximum. L'objectif est de trouver un ordonnancement minimisant le makespan.

Nous proposons des résultats optimaux de complexité et des algorithmes pour des

cas simples. Pour des tâches de taille arbitraire, le problème peut être résolu optimalement par un algorithme glouton quand tout les chaînes ont une longueur 1. À partir du moment ou une application est une chaîne de longueur au moins deux, le problème est NP-complet. Quand toutes tâches composants les applications ont les mêmes durées, nous proposons un algorithme "round-robin hiérarchique" et prouvons son optimalité.

Nous appliquons ensuite ce même algorithme dans le cas d'applications periodiques ayant le même nombre n de période et montrons qu'il constitue une $1 + 1/n$ approximation du résultat optimal et que cette borne est atteinte.

Enfin nous proposons une résolution approchées dans le cas général. Nous démontrons que les heuristiques de liste fournissent une 2-approximation du résultat optimal peu importe l'ordre utilisé.

En pratique, l'intuition nous laisse penser que l'ordre utilisé par les heuristiques de liste a un impact et qu'il y a une différence entre leur performance réelle et théoriques.

Afin d'étudier cette différence, nous simulons leur comportement dans différents scénarios dont la présence de bruit faussant la durée espérée des phases.

Contrôle de ressources par le placement d'applications

Après avoir considéré le placement des applications donné afin d'étudier l'ordonnancement des opérations d'E/S, nous avons voulu, dans une seconde partie, élargir le problème et fournir un algorithme fournissant ce placement tout en prenant en compte les requêtes de bande passante.

Cette partie permet aussi de développer notre méthode de travail: construire des modèles théoriques simplifiés basé sur l'observation de systèmes réels, construire des solutions basées sur ceci pour enfin les évaluer dans des contextes réalistes.

Nous avons étendu le modèle de machine présenté précédemment afin d'inclure à la fois les besoins en terme de calcul et en terme de bande passante des différentes applications. Nous avons commencé par démontrer par l'expérience que les simples heuristiques de listes "équitables" exhibaient les meilleurs performances pour l'ordonnancement des opérations d'E/S. Puis, nous avons cherché à définir quels groupes d'applications vont être exécutées de manière concurrente sur la plateforme. Pour cela, nous avons défini des groupes d'applications en vérifiant d'une part la disponibilité des ressources de calcul nécessaire mais aussi celle de la bande passante. Pour vérifier facilement ces contraintes, le comportement périodique des applications en terme d'E/S est approché par une quantité moyenne d'occupation de la bande passante. Concrètement, après leur avoir associé cette valeur, les applications sont triées selon leur temps espéré d'exécution (afin d'éviter une sous utilisation des ressources de calcul en fin de paquet, nous discuterons d'autres ordre plus loin) puis l'on place chaque application dans un paquet selon une politique « meilleur correspondant », un nouveau paquet est créé si aucun ne convient. Chacun est ensuite exécuté séquentiellement. Dans le cadre d'une collaboration avec l'Université Carlos III de Madrid, nous évaluons les

performances de cet algorithme en l'implantant dans leur middleware CLARISSE et conduisant des expériences par émulation avec celui-ci sur un de leur calculateur.

Avec un seul nœud d'E/S, Nous démontrons un surcoût en terme de temps d'exécution comparé à des algorithmes ignorant les E/S dû à un plus grand contrôle des ressources. Cependant, nous observons une baisse significative du temps (imprévisible) perdu par le système lors de la contention au profit d'une inactivité (définie et prévisible). Ces résultats sont permettent d'une part d'espérer de meilleures performances en cas politique de récupération (remblayage). D'autre part, quand nous simulons une augmentation du nombre de nœuds disponibles et permettons l'exécution parallèle de plusieurs paquets, notre approche garanti à la fois un meilleur makespan et une meilleure équité entre les utilisateurs.

Comparaison des modes de contrôle

L'algorithme par paquets défini dans la partie précédente impose que les sous-ensemble d'applications exécutés en parallèle aient le même début d'exécution. Dans cette partie, nous cherchons à le comparer de manière qualitative à une autre politique classique de gestion les ressources d'une plateforme: les heuristiques de listes. Ces deux stratégies sont plébiscité par les administrateurs de systèmes de calcul en raison de leur simplicité à implémenter et tester ainsi que leur faible complexité qui permet de les appliquer à grande échelle. Par ailleurs, elles représentent toutes deux différentes manières d'appliquer les contraintes en terme de ressources. En définissant groupant les applications à exécuter en paquets partageant une même date de début d'exécution et dont la fin est définie de manière fiable, l'algorithme par paquet définit une fenêtre de temps dans laquelle la contrainte de bande passante peut être dépassée à condition qu'elle soit respectée en moyenne. Un algorithme de liste n'impose pas une telle contraints: il trie les différentes applications selon un ordre de priorité (typiquement par ordre d'arrivé dans les ordonnanceurs de tâches). En contrepartie d'une plus grande liberté sur les dates de début d'exécution, la contrainte de bande passante doit donc être respectée à tout instant. Pour comparer ces deux paradigmes nous avons voulu conserver les propriétés définies dans la partie précédentes: les stratégies se basent sur des valeurs moyennes pouvant être collectées par des outils comme Darshan et peuvent être implémentées dans un middleware qui gère les E/S en direct. Afin de conduire notre évaluation expérimentale, nous avons implémenté un simulateur afin d'évaluer ces deux approches dans plusieurs scenarios inspiré de la littérature relative à des supercalculateur. Nous observons des performances variants fortement selon le profil de la charge de travail sous-jacente. Bien que les ordonnancements par listes soient généralement les stratégies d'allocations standard dans les ordonnanceurs et fonctionnent particulièrement bien avec des jeux d'applications hétérogènes, les jeux d'applications uniformes et gourmands en E/S peuvent tirer partie des stratégies par paquets. De plus, construire des paquets d'applications basés sur les temps caractéristiques permet, sous des condi-

tions précises d'homogénéité (génération uniforme, répartition des I/O suivant une loi normale) et de charge en E/S, une synchronisation des opérations améliorant grandement la performance par rapport aux autres stratégies. Nous soulignons par ailleurs la relation forte entre les caractéristiques des applications constituant la charge de travail et la pertinence de la stratégie d'ordonnancement et fournissons des pistes sur comment choisir la plus approprié.

Prospective

Les résultats de la partie précédente nous invite à réfléchir à des stratégies hybrides. En regroupant les applications similaires et en partageant la bande passante entre de tels sous-ensemble d'applications similaire, nous espérons tirer le meilleur des deux stratégies Dans ce manuscrit, nous présentons des exemple minimaux montrant que des applications similaires tirent bénéfice d'accès exclusifs à la bande passante alors que des applications très différentes préfèrent le partage. Nous fournissons ensuite un modèle mathématique minimum montrant que regrouper des applications ayant la même période n'impacte que marginalement l'équité entre les utilisateurs. Nous approfondissons cette direction avec des objectifs doubles: définir les groupes d'applications devant être en concurrence et déterminer quelle proportion de la bande passante allouer à chaque groupe. Une première intuition sur laquelle nous travaillons serait de grouper les applications en s'intéressant seulement à la longueur de leur itérations et allouer plus de bande passante à celles ayant des phases courtes et une grande fréquence d'E/S. Nous cherchons à répondre à ces questions en mettant en place un système d'évaluation réaliste utilisant SimGrid.

Abstract

In high performance computing, platforms (the supercomputers) are composed of computational resources divided into racks. Each rack have one (sometimes several) I/O nodes to access the parallel file system (PFS). Following the growth and the optimization of the machines, the amount of data involved in the calculation is increasing. As different parts of these racks are allocated to independent applications, and as the storage is shared between these, too many concurrent accesses can lead to contention and performance loss.

This thesis consist in studying strategies to map applications on the platform and scheduling the different accesses to the storage system in order to avoid contention. We discuss the relevance of our solution both from the administrators point of view (using the throughput) and from the users one (using stretch).

In a first part, we assume that the mapping was done upstream and only study the scheduling of bandwidth accesses in order to minimize the latency. We show the intractability of the standard I/O scheduling problem, discuss some simple cases and show the relevance of List-scheduling policies.

In a second part, we extend our problematic to the problem of mapping applications into the nodes in order to avoid concurrent accesses.

To solve this problem, we propose a pack algorithm which, on a single I/O node, increases greatly the stretch while slightly degrading the makespan. However, this degradation in the consequence of a stricter control on resource usage. This allows to a better scaling with more I/O nodes as well as facilitating loss recovery with back-filling for example.

Finally, we discuss how to extend standard resource management strategies such as list scheduling and the aforementioned pack scheduling in order to take I/O into account and what it means in terms of resource utilization and performance. Such strategies are largely used by HPC platform for resource allocation due to their simplicity. We observe that the pack scheduling policy induces a limitation on average that enables intensive I/O phases on condition that they are compensated during the execution. In comparison, list strategies always follow the prescribed restriction and are therefore more restrictive. It follows that pack scheduling are to be preferred when high quantities of data compared to the available bandwidth are involved.

In most of our work, we considered that I/O accesses were exclusive. However, in highlighting notable performance discrepancies depending on the similarity between applications, we showed that some should be executed in parallel. As a conclusion, we discuss the recent development of our models seek to determine how to define these applications partitions.

Chapter I

Introduction

I.A Introduction

I.A.1 A quick presentation of HPC

High Performance Computing (HPC) is the scientific framework around large-scale computational platforms called supercomputers. Nowadays, scientists rely on these machines to run programs that could not run on simple computer. Their scale can be dizzying: the last journey [35], an astronomy simulation project carried out on Mira, the Argonne Leadership Computing Facility supercomputer evolved approximately 1.24 trillion particles producing almost 150 TB of data. Running voracious applications is not a cosmologist privilege, climatologists and oceanographers complete their field observations using simulation. Modeling also bears promising results in medical science [37]. This programs are often very complex and composed of multiples specific tasks with precedence constraints.

In this aspect, supercomputers must be able to handle diverse requirements coming from concurrent users independent to each other. Indeed, the building and maintaining cost of these mastodons can only be justified with a diverse and efficient usage. Aurora [4], an exascale machine under development at Argonne was estimated at more than $500M With a power consumption of 30 to 40 MW, the current leader of the top 500 supercomputer, the Japanese Fugaku [5] consume several time the power output of a little nation (Togo: 10.3 M). New problems emerges from these extreme orders of magnitude opening new theoretical and optimization problems.

I.A.2 Evolution

Historically, as it became harder to add more and more component to a single module. Cray decided in the 70s to split their computer into multiples modules, designing the first supercomputer.

Nowadays supercomputers are composed of millions of cores, reaching the exaFLOP/s (10^{18} floating-point operations per second), the equivalent of ten million per-

sonal computers (100GFLOP/s).

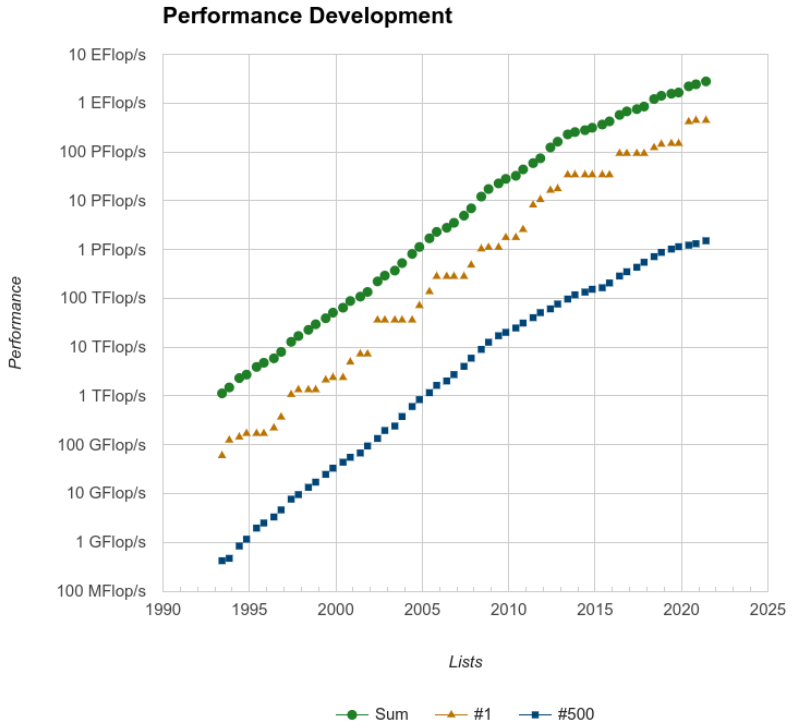


Figure I.1: Performance development of Top500 supercomputers (source: https://www.top500.org/)

In the mean time, Input/Output (I/O) throughput evolved but failed to keep up the pace. A statement recalled recently by Khan et al. [42]. As shown by successive supercomputer at Argonne: Mira (2013 - Peak performance: 10 PFlop/s; peak I/O throughput: 240 GB/s) is an upgrade of the older Intrepid (Peak performance: 0.56 PFlop/s; peak I/O throughput: 88 GB/s). Yet, the next generation calculator of the same facility, Aurora, expecting to reach the ExaFlop/s, only offers a storage of 25 TB/s per switch [4]. I/O throughput still scaling linearly to the previous one.

This is especially crucial as a linear increase of platform capacity does not leads to an equivalent increase in performance.

The recent development of data-intensive computing applications such as high-performance data analytics (HPDA) and deep learning (DL) contributes to shift the computer-centric point of view to a data-aware approach. The growing demand for data processing is accompanied by disruptive technological progress of the underlying storage technologies. As a result, upcoming exascale HPC systems are transitioning from a simple HPC storage architecture, consisting of a parallel back-end file system and

archives often based on tapes, towards a multi-tier storage hierarchy that includes Non-volatile main memory (NVMM) with a performance close to DRAM, NVMM-based SSDs inside compute nodes with a bandwidth of many GB/s, SSDs on I/O nodes, parallel file systems, campaign storage, and archival storage.

I.A.3 The IO bottleneck

> *"Very few large scale applications of practical importance are not data intensive"* - Alok Choudhary, Apr. 2012

Despite this architectural shift, the usage of the I/O stack is not optimal since end users lack the information about the state of the HPC resources and the I/O accesses of the multiple applications running in a supercomputer. As a result, opportunities for global I/O optimization are missed mostly due to uncoordinated data management, which often leads to redundant data movement, large I/O accesses contention and delayed end-to-end performance.

In some scenario, sole applications can saturate a cluster bandwidth. As an example, despite various optimizations, the aforementioned HACC simulation only reached 56% of its ideal 26.7 GiB/s peak bandwidth [33].

Parker et al. [57] make an evaluation of the 'Theta' platform in Argonne. Using application as benchmark, they show that performance depended heavily on the underlying architecture. The LAMMPS application attaining a peaked memory bandwidth ranging from 60 to 170 GB/s depending on the resources used.

Concurrent applications sharing larger machines leads to an increased risk of I/O interference [80, 65]. Concurrent accesses are often performed in "best-effort" mode leading to contention. The importance of this phenomenon is still discussed. It is unclear whether the amount of performed operation is degraded when several applications compete. Some studies tend to show that contention is *over-additive* (due to hardware restrictions, the time spent by each application executed simultaneously is larger than the time that each would spend without contention if they were executed alone). Other studies failed to show the existence of a performance loss worse than linear. However, it is straightforward to notice that in both cases, concurrent accesses leads to delays for the application waiting or having only partial bandwidth usage. This would then reverberate throughout the execution

In this context, it is more important than ever to include data in our way of thinking, designing and running HPC platforms.

I.A.4 Motivational example

Cross-application contention has been studied recently [34, 65, 74]. The Hashimoto and Haida study [34] evaluates the performance degradation in each application program when Virtual Machines (VMs) are executing two application programs concurrently in a physical computing server. The experimental results indicate that the interference

among VMs executing two HPC application programs with high memory usage and high network I/O in the physical computing server significantly degrades application performance. A 2005 study by Skinnet and Kramer [65] cites application interference as one of the main problems facing the HPC community. While the authors propose ways of gaining performance by reducing variability, minimizing application interference is still left open. A more general study by Xie et al. [77] analyzes the behavior of the center-wide shared Lustre parallel file system on the Jaguar supercomputer and its performance variability. One of the performance degradations seen on Jaguar was caused by concurrent applications sharing the filesystem. All these studies highlight the impact of having application interference on HPC systems, without, but they do not offer a solution.

As a way to forge our own understanding of the phenomenon , we evaluated the inter-application interference in PlaFRIM cluster [2] cluster for three representative use cases. Each compute node of PlaFRIM consists of two 12-core Intel Xeon E5-2680 processors and 128GB RAM. The nodes are connected by Infiniband QDR TrueScale at 40Gb/s, and the filesytem is Lustre configured with one metadata server, and four object storage servers. We evaluated the system I/O performance by executing use-cases under two different configurations: single application and multiple applications. In single application one program was executed exclusively, having full access to complete I/O bandwidth provided by the system. In contrast, with multiple applications, several program instances are executed simultaneously (accessing to different files) and competing for the I/O resources.

Figure I.2 shows the I/O bandwidths of each use-case and configurations. Use cases A and B corresponds to an MPI program that writes a distributed matrix in a file using non-collective calls. More precisely, in use case A, each process writes the data consecutively, whereas in use case B a striped write access is performed with a stride size of 195 MB. Use case C is the NAS BTIO simple benchmark, which also uses non-collective MPI calls for accessing the file. In this use-case the I/O has a reduced data granularity. Because of this, use case C has a smaller I/O bandwidth than the other counterparts.

Note that for all the use cases the I/O bandwidth degrades when multiple applications are executed. We define this degradation d as the percentage of bandwidth that is lost when multiple applications are being executed. More formally, $d = 1 - \frac{\sum BW_{multi}}{BW_{single}}$ where BW_{single} is the I/O bandwidth for single application configuration and $\sum BW_{multi}$ is the aggregated bandwidth for multiple application configuration. In our experiments we ran 4 applications are the same time and we saw a, degradations of 2%, 16% and 11%, for use cases A, B and C, respectively. There are many reasons [80] for this degradation (that does not always occur) that are related to the complex interaction between the applications and all the levels of the I/O subsystem. One of the main factors occurs when the storage servers receive requests from multiple applications: One is that it is necessary to access to more locations in the hard disk surfaces, reducing the access locality and I/O performance; Another reason is the contention at I/O node-level, given that the raid servers are potentially connected with all the I/O

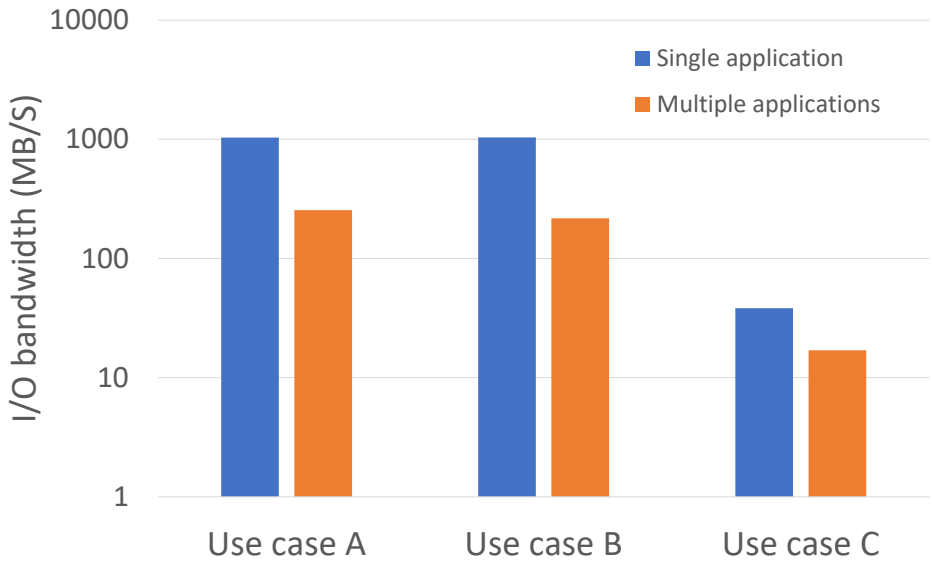


Figure I.2: I/O bandwidth comparison for single and multiple applications of the three use cases. In use cases A and B an MPI program that writes a distributed matrix in a file using non-collective calls. Each application uses 96 processes for writing a 47.7GB file. With multiple applications, four instances of the same application are executed simultaneously. Use case C corresponds to the NAS BTIO simple benchmark that operates with an 1.6GB file using 64 processes. With multiple applications, two instances of the same application are executed simultaneously.

nodes.

Based on these results we conclude that in some contexts, simultaneous inter-application I/O should be avoided. This is the idea behind the CLARISSE I/O scheduling control protocol [39]. Using a publish-subscribe protocol, when multiple applications access to the disk, only one is granted to perform the I/O. The remaining ones are delayed waiting for the I/O completion.

I.B Context

I.B.1 Existing solutions

The issue of conflicting accesses on the I/O subsystem is a well known problem in HPC infrastructures [23, 80, 58]. In the current section, we present some of the approaches

aiming at tackling the problem.

I.B.1.a Data transformation

Recent studies propose application-side strategies based on I/O management and transformation. As contention arises with large amount of data, recent studies propose application-side strategies based on I/O management and transformation. Lofstead et al. [53] study adaptive strategies to deal with I/O variability due to congestion by modifying at certain times both the number of processes sending data, and the size of the data being sent. Tessier and al. [71] focus on the locality of aggregate nodes. These nodes are compute nodes dedicated to accumulate data sent by other compute nodes during the I/O phase of an application. Those nodes also have the possibility to transform the data being sent (for instance by compressing it [22]). To go further, data can even be compressed in a lossy way [20]. In-situ/intransit analysis developed in recent works [25] try to deal with file systems reaching their limit. In the past, some workflows used to create the data and to store it on disks before analyzing it as a second step. In-situ/in-transit analysis offers to dedicate some specific nodes to the analysis and to perform it as the data is created. The goal is to reduce the load on the file systems. We consider that all these solutions occur uphill to our problem and hence can be used conjointly.

I.B.1.b Software to deal with I/O movement

Auto tuning Work using machine learning for auto tuning and performance study [11, 46] can be applied for I/O scheduling but do not provide a global view of the I/O requirements of the application. Coupling with a platform level I/O management ensure better results. Isakov et al. [40] propose to design explainable models using machine learning. They emphasize on the gap between black box model providing accurate predictions and less precise but simple and intuitive models.

Middlewares Singh et al. [63] present a strategy that combines I/O conflict prediction and application malleability. By means of prediction, the future I/O conflicts are forecast. Based on that, the time of the next I/O phase is shifted in order to avoid the I/O conflict. This is carried on by dynamically changing the number of processes of the conflicting application (by means of malleability). AHPIOS [38] is a light-weight ad-hoc parallel I/O system with elastic partitions that can scale up and down with the number of storage resources. Other works such as those of Lim et al. [50] or Cheng et al. [18] present elastic solutions using Hadoop Distributed File System (HDFS). SpringFS [79] presents an elastic filesystem for Cloud computing platforms.

The middleware CLARISSE [39] proposes mechanisms for designing and implementing cross-layer optimizations of the I/O software stack. The specific implementation of the problem considered here is a naive First Come First Served approach. They, however, provide an excellent opportunity to study our results in a real framework.

Online schedulers Finally, the solution closest to our approach is to schedule the I/O of applications in order to mitigate interference. Several works have tackled this problem. Some approaches [29, 82] use priority function scheduling at the I/O node level: they consider applications already mapped on a machine, sharing some I/O bandwidth. Several priority functions have been proposed by Gainaru et al. [29] to optimize either the equity between applications or the machine throughput. Another recent approach developed by Aupy et al. [9] is to look at structural properties of the applications (such as a periodic behavior in I/O communications) to develop more advanced schedules. In [23], the authors investigate the interference of two applications and analyze the benefits of interrupting or delaying either one in order to avoid congestion. Unfortunately their approach cannot be used for more than two applications. Another main difference with our previous work is the light-weight approach of this study where the computation is only done once. In TWINS [12] the access to the I/O nodes is coordinated at the I/O forwarding layer, reducing the contention. Another similar approach is AS-CAR [49], that uses traffic controllers on storage clients to detect I/O congestion and introduces traffic rules to reduce it. AIS [52] is an I/O-aware scheduler that performs an offline analysis of the I/O traffic for identifying I/O characteristics of the applications. Based on that, the applications are scheduled avoiding I/O conflicts. Other works that use models to predict and avoid the application I/O interference are [6, 67, 21].

Batch scheduler There have been some recent works trying to incorporate several dimensions (such as I/O needs and compute nodes) into batch schedulers. On the more theoretical side, several work have included these components to a scheduling problem [32, 73]. Bleuse et al. [13] have considered geometrical constraints where, given some network topologies, they try to schedule applications while respecting their request both on the number of compute nodes and on the number of dedicated I/O nodes. To the best of our knowledge, there is no specific work to consider the scheduling of compute nodes while considering possible interference on I/O. On the practical side, Herbein et al. [36] studied incorporating I/O awareness in batch scheduler strategies. To develop a solution, they considered a simple model where all jobs had an I/O need proportional to the number of nodes needed. Their observation was that taking I/O into account reduced job performance variability.

I.B.1.c Hardware solutions

Diminishing I/O bottleneck can also be thought at the architectural level. A previous paper [51] noticed that congestion occurs on a short period of time and the bandwidth to the storage is often underutilized. As the computation power used to increase faster than the I/O bandwidth, this observation may not hold in the future. In the meantime, delaying accesses to the system storage can smoothen the I/O request over time and tackle latency. An example of this technique is presented in Kougkas et al [45]. A dynamic I/O scheduling at the application level, using burst buffers, stages I/O and allows computations to continue uninterrupted.

They design different strategies to mitigate I/O interference, including partitioning the PFS, which reduces the effective bandwidth non-linearly. Note that for now, these strategies are designed for only two applications, furthermore they are not coupled with an efficient I/O bandwidth scheduling strategy and can only work because they considered an underutilized I/O bandwidth.

Sizing strategies have been studied [7], as well as buffer placement (shared or distributed) [8, 43]. Tang et al. [70] have studied draining strategies and have shown that the natural reactive strategy to empty the buffer as soon as possible can lead to severe degradation. Aupy et al. [7] have shown theoretically that only emptying the buffer when it is at least 15% full do not lead to significant delays compared to the reactive strategy, however it may mitigate the issues raised by the work of Tang et al. [70]. Finally, as was shown in the recent work by Aupy et al. [8], to be efficiently used the buffers still need to be coupled with I/O management strategies.

Boito et al [14] design multi-layer memory architectures to mitigate the congestion occurring at the I/O bandwidth level. The architectural enhancements are aiming to fluidize I/O requests. It is often accompanied by the development of data middleware, such as the design of an intermediate aggregating layer which enables collective operations [72, 64]

I.B.1.d Related theoretical problems

This overview of approaches to tackle the I/O bottleneck would not be complete without mentioning classical theoretical problem related to scheduling with different resources. Indeed, on the application side, if we choose to transform neither the execution nor the output, the I/O contention issue is treated as scheduling problem [53, 81]. The MS-HPC-IO problem of section II.A may recall the classical job shop problem (a optimization problem where jobs needing to be schedule on different machines with - see definition in [48]). In both problems jobs are composed of dependent tasks that have to be performed on specific machines. However, here, we do not have constraints on the computation machine therefore if knowledge of job shop can help to develop insight of solutions, it can not be used straightforwardly for HPC-IO. Variants of job shop and flow shop are abundantly discussed in the literature: [47, 48, 69, 15]. We recall that flow shop is a particular case of job shop where the operation sequences do not depend on jobs.

I.B.2 Monitoring

Monitoring softwares usually capture I/O accesses from HPC application execution in order to be analysed to provide system and application characterization. However, in order to be effective they need to capture the appropriate data for an expert to build models and heuristics leading to performance improvement. In practice, monitoring tools are developed using system-level counters either at an application level or at a

platform level. This makes a lot of sense for the engineers and the system optimization. However, when we need to broaden the scope and have a global vision of the current behavior of applications it often falls short. It is indeed hard to capture the individual pattern of applications when monitoring the platform or the interaction with other applications running when monitoring an application execution. Nonetheless the data they provide is invaluable in order to build models. Tools commonly used include TAU [62], Paraver [60], SCALASCA [31], Paradyn [55], Darshan [16, 66]. Darshan is especially present in both the current thesis and related work from the same field as it is specialized into capturing I/O traces.

I.B.3 Application modeling

I.B.3.a Periodic behavior

Observations shows that many HPC applications [16, 24, 29] periodically alternate between (i) operations (computations, local data accesses) executed on the compute nodes, and(ii) I/O transfers of data. As shown in these studies, this behavior can be predicted beforehand. Figure I.3 presents an example of I/O behavior. Data was provided by ATOS and shows pseudo-periodical I/O operations.

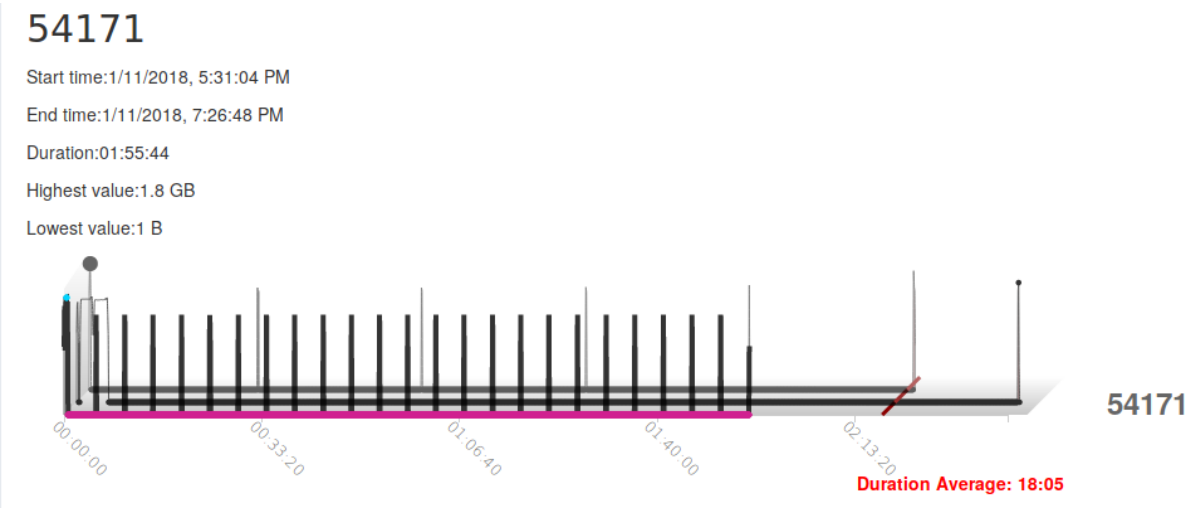


Figure I.3: Roughly periodical I/O behavior of an application, obtained from data provided by ATOS

I.B.3.b I/O prediction

Literature discussing the I/O behavior prediction is abundant and convincing. Oly and Reed [56] use I/O traces from scientific code to build Markov models representing the spatial patterns of such applications.

Dorier et al. [24] construct a grammar based model in order to predict the occurrence and amount of data of future I/O operations.

Other approaches to predict I/O are large scale log analysis [44] or machine learning approaches [54].

I.C Positioning

HPC is a large domain regrouping different actors who have all their own objectives. There are engineers and administrators aiming to optimize or design their platforms to answer their users requests. There are users who want their applications to run as smoothly as possible on the platform available to them. On the opposite side of the spectrum, there are theoretical algorithms aiming to have results the most advanced theoretically even if it means using oversimplified models. We want to be in between. That is having the global vision and the force of abstraction of theoretician in order to build the necessary knowledge to design well-thought future systems. On the other hand, we want our solution to be in touch with the current reality of our field. Figure I.4 present schematically our approach. We take observations made on supercomputers as a starting point to design realistic models. This model might be too complicated to build theoretical proofs or design precise algorithms but is used as a thinking basis. Then, we reduce it in order to have a simple version with few parameters. This model simplicity serving to define clear theoretical objectives and to help designing strategies accordingly. Finally, we test the strategies on the realistic models and study the discrepancies in order to raise more concern and refine our models.

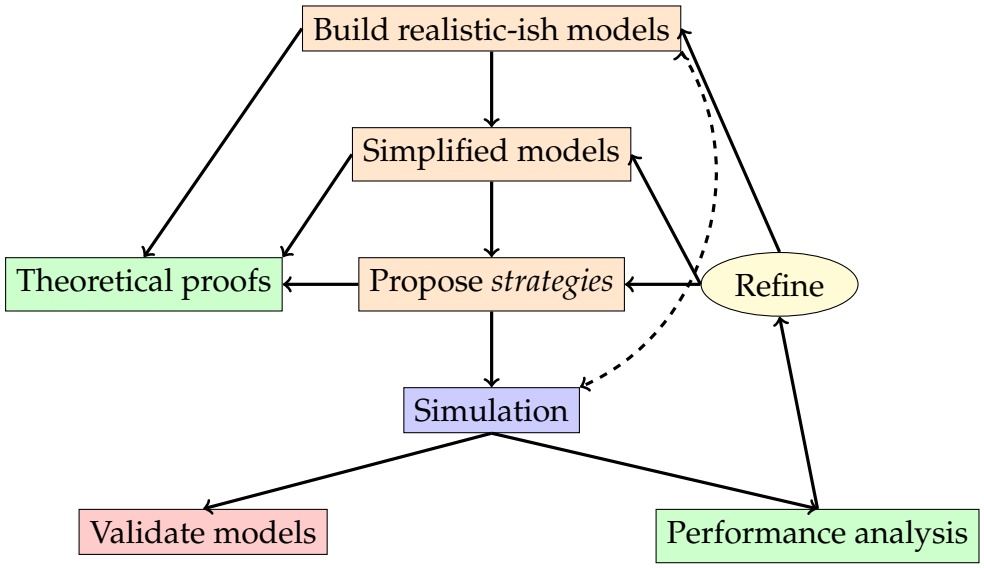


Figure I.4: Simplified vision of our approach. Contributions are presented in orange

I.C.1 Simulating HPC workloads

I.C.1.a Scientific context

It seems pertinent to discuss the importance of simulation in the current manuscript. The unfolding of the present thesis follows more or less the chronological path of its development. we chose to keep it as such to follow the scientific progress of my studies. Notably this last year we took a large step back with regards to experimental approach.

In chapter II, we settled for an ad-hoc python code aiming to model the problem in the most direct way. This technique may be the most time efficient but it lacks in re-usability and cannot provide insight outside of its limited scope. The experiments performed for chapter III were run on Tucan, a cluster based in Charles III University of Madrid and maintained by the arcos research group [1] and using an emulator of the software Clarisse. Here, we only modified a small dedicated subpart of a tool developed independently: the scheduling heuristics. we could also choose and generate the application profiles used by the emulator. Thus our experiments were much more constrained with an impact on what could be done as policies but with more "realistic" executions. Or at least, experiments closer to what it is possible to implement in a real cluster with minimal engineering skills.

The European development of the pandemic early 2020 slowed down international collaborations. As universities and labs closed down and as scientists juggled more that ever between their teachings, the scientific challenge and their own personal life, it became harder to work conjointly on a same software or in the same platform. In this context, we decided to implement our own simulator.

I.C.1.b Implementation of the simulator

We attempted to produce an application divided in complementary yet roughly independent component.

1. A workload generation module
2. A module defining the application schedule
3. A module scheduling the I/O accesses
4. A module emulating the platform

Each parts uses the result of the previous one. However, we expect to implement different approaches at each level. This way we can keep the same execution stack and study the impact of only one of this stage by tuning the adequate component.

Generating applications How to generate applications is crucial in performance evaluation. [27, 26] In a well-written article, Feitelson [26] goes through several pitfalls to avoid when performing experiments on workloads In the introduction of his book [26],

he recalls that in the literature, almost all published articles exhibits results outperforming the state of the art. His interpretation is not that scientist are lying about performance but rather that the way the experiments are performed and mainly the way workloads are simulated and evaluated are the product of presupposed assumption and impact the scope of the experiment. In that case, it is capital to make clear and explicit assumption on the workload at hand and discuss the possibilities and the limitation of each experimental setup. Hence, in this thesis, the nature of HPC workloads and their modeling is a constant concern. In most of the experiment, we chose to generate applications in order to support our theoretical models therefore sacrificing realism in order to explore more extensively the parameters space. However, upstream of the work presented in Chapter IV, we saw the limitation of this simulation scheme and chose to implement a workload simulator based on the available data from Mira [59]. Details about the generation is provided in the related experimental sections.

Resolve schedules We chose to implement a set of queues one for each status an application can have while running on a platform (waiting for a given resource, performing I/O, computing...). They are initialized based on the application states. In order to emulate the platform, we compute the next event occurring and update both the applications and the queues states.

All sources from this simulator are available on my gitlab repository [**?**] alongside simulation scripts for chapter II.

I.C.2 Contributions

In Chapter II, we present a simple model for I/O scheduling focusing solely on the I/O operations. We discuss the complexity of the problem at hand and provide theoretical solutions for simple scenario including list-scheduling strategies and uniform tasks. Then, we experiment heuristics based on the periodical behavior of applications and study their robustness.

We extend this model in Chapter III by taking into account the mapping problem that we left aside previously. In order to perform I/O aware allocation of the platform, we define an algorithm based on packs. We show that, compared to a baseline, the I/O aware pack algorithm sacrifices overall machine throughput in favor of a better control of the time loss in contention and a better fairness. This increased control on resource usage enables a better scaling when increasing the platform size as defined by racks (batches of compute nodes associated to an I/O node).

In Chapter IV, we compare the pack strategy aforementioned with the simple list scheduling strategy with I/O awareness. These two strategies are used as tools to understand and discuss the different ways to enforce bandwidth control and the impact of workloads on the performance of said control. We showed that choosing a strategy must depend on workload characterization, mainly the I/O intensity of the applications and their similarity.

In the conclusive Chapter V, we make a retrospective of the last year in regard to scientific path and progress. Finally, we open up on some ongoing research: based on the observations of Chapter IV and some simple examples, we try to define groups of applications should perform I/O exclusively while sharing bandwidth with others.

Chapter II

Scheduling bandwidth accesses

In the previous chapter, we presented the I/O bottleneck problem. In HPC settings, the application I/O pattern leads to contention on the filesystem and to a loss of performance. We ought to propose strategies to handle the I/O operations while limiting this issue. Aupy et al. [9] design a strategy which uses the periodic nature of HPC application to develop efficient periodic scheduling strategies for their I/O transfer. We take this work as an entry point to address the I/O bottleneck. Then, as our main objectives is to define scheduling algorithms. From their focus on the applications and their phase-based behavior, it follows two problems.

- How do we model formally the HPC applications behavior in our context?
- How do we schedule the I/O accesses?

This chapter deal with these two subjects. First we propose a mathematical model for HPC applications. We emphasize on the periodic behavior of most. Then, we design static algorithms that prescribe when each application can access the storage. These two basic problems are crucial for all of the remaining thesis. The mathematical model we define here will be used, with adequate modifications, as a basis for future problem definitions. Then, when increasing the scope and dealing with applications mapping and different strategies, we will still need to schedule I/O calls at the lower level and use the subsequent scheduling policies.

In the following, we first define formally the application model and the optimization problem (Section II.A). Then, we provide theoretical result from the literature and polynomial algorithms for simple cases in Section II.B. In Section II.C we discuss approximation for the general scheme. These algorithms are then experimentally evaluated using simulation in Section II.D.

II.A Bi-colored chains model

II.A.1 Machine model

We consider a platform consisting of two types of machines: type $\mathcal{A}$ and type $\mathcal{B}$. Each of these machines can have either a bounded number of resources or an unbounded one as would be the case in a typical scheduling problem. In the I/O problem under consideration in this chapter, we consider that the jobs are already scheduled on the compute nodes (machine of type $\mathcal{A}$) and that there is no competition at this level. Hence, we can assume without loss of generality and unbounded number of such resources. On the contrary the bandwidth of the Parallel File System (PFS) (machine of type $\mathcal{B}$) is shared amongst the different jobs. Hence, we say that it has a bounded number of resources B. For the sake of simplicity, we normalize the bandwidth usage and consider B =1. We call this instance of the platform an *bi-colored platform*.

Figure II.1 shows an overview of this model and jobs executed on this platform.

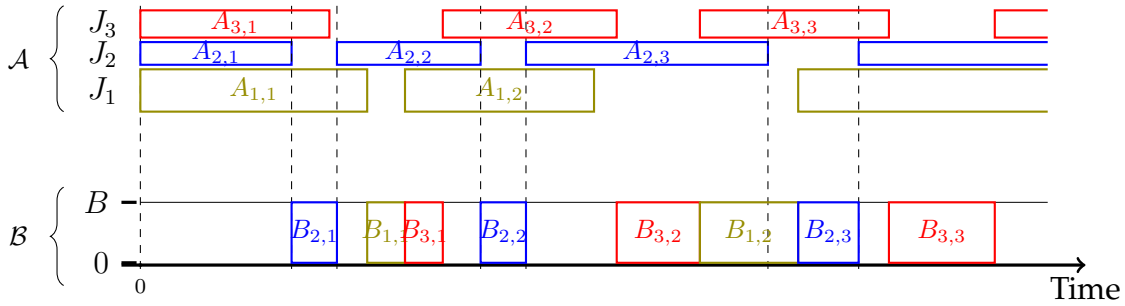


Figure II.1: Schematic overview of three jobs J_1, J_2, J_3 scheduled on a bi-colored platform.

II.A.2 Job Model

We consider scientific applications running simultaneously onto a parallel platform [9, 8]. The set of processing resources are already allocated to each application. With respect to I/O, applications consist of consecutive *non-overlapping* phases: (i) a compute phase (executed on machine $\mathcal{A}$); (ii) an I/O phase (executed on machine $\mathcal{B}$) which can be either reads or writes.

Formally, a job J_i consists of $2n_i$ successive operations $A_{i,j}$,$B_{i,j}$, ($j \leqslant n_i$). The dependencies that need to be respected are such that: $A_{i,j+1}$ (resp. $B_{i,j}$) can only start its work when operation $B_{i,j}$ (resp. $A_{i,j}$) is done entirely. We denote by $a_{i,j}$ (resp. $b_{i,j}$) the volume of work of operations $A_{i,j}$ (resp. $B_{i,j}$). In the current problem, because there is no constraint on the number of compute nodes allocated to J_i, we can assume without loss of generality that it is equal to 1 and $a_{i,j}$ also corresponds to the execution time of operation $A_{i,j}$. Similarly, when $B_{i,j}$ uses the full I/O bandwidth ($B = 1$), $b_{i,j}$ corresponds to the minimal time to execute operation $B_{i,j}$.

We call such jobs *bi-colored chains* and write them:

$$J_i = \prod_{j=1}^{n_i} (A_{i,j}, B_{i,j}) \tag{II.1}$$

The minimal execution time of J_i is given by the equation:

$$C_i^{\min} = \sum_{j=1}^{n_i} a_{i,j} + b_{i,j} \tag{II.2}$$

In addition, in this chapter we consider some specific jobs called *Periodic* jobs. They consist in successions of identical (in volume and time) compute operations and I/O operations. Those are typical patterns in High Performance Computing [16, 29, 24]. We extend the notation for bi-colored chains to these jobs:

$$J_i = ((A_i, B_i)^{n_i}) \tag{II.3}$$

Remark and motivational example While in real-life, platforms can allow bandwidth sharing especially *fair-share*, this model describes *exclusive I/O accesses*. This simple choice can be explained by a minimalist example described in Figure II.2. Two applications performing a compute phase and an I/O phase. The total execution time is the same. However, when using exclusive accesses, App. 1 terminates earlier therefore providing a better user experience as well as a earlier resource availability.

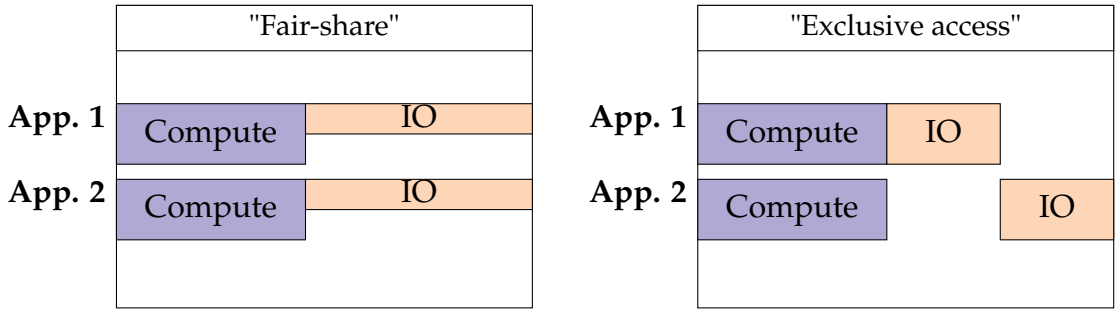


Figure II.2: Comparison of "Fair-share" and "Exclusive accesses" on a single iteration

II.A.3 Optimization problem

In this context, we define the MS-HPC-IO optimization problem. We consider the specific model where the I/O of tasks is rigid: for all applications, the I/O is always performed at full bandwidth and cannot be pre-empted. This model is what is currently implemented in Clarisse [39].

A schedule $\mathcal{S}$ is fully defined by giving an order for the different I/O operations on the machine of type $\mathcal{B}$. Indeed, because there is no competition for the resources of type $\mathcal{A}$:

- $A_{i,1}$ can start immediately;
- $B_{i,j}$ can start as soon as both events are finished: (i) $A_{i,j}$ is finished; (ii) all jobs anterior to $B_{i,j}$ in the schedule on the machine of type $\mathcal{B}$ are finished.
- $A_{i,j+1}$ can start as soon as $B_{i,j}$ is finished.

Hence, we can formally define a schedule in Definition 1

Definition 1 (A schedule $\mathcal{S}$)**.** Given a set of jobs $J_i = (\prod_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$, a schedule $\mathcal{S}$ is defined by a permutation of the jobs $((B_{i,j})_{j\leq n_i})_i$ that satisfies, for all i, j, $B_{i,j}$ is before $B_{i,j+1}$

We consider the classical objective function for scheduling problem. It corresponds to the system performance (makespan or execution time). In the future, we may study system fairness as well.

Let C_i be the end of the execution of a job J_i in the schedule $\mathcal{S}$. We define the makespan $C^{\mathcal{S}}_{\max}$ of the schedule $\mathcal{S}$ to be:

$$C^{\mathcal{S}}_{\max} = \max C_i \tag{II.4}$$

Definition 2 (MS-HPC-IO)**.** Given a set of rigid bi-colored chains $J_i = (\prod_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$, and an I/O platform. Find a schedule that minimizes the makespan $C^{\mathcal{S}}_{\max}$.

II.B Complexity of the MS-HPC-IO problem

II.B.1 Intractability

In this part, we briefly present some intractability results from the literature for MS-HPC-IO.

In the literature, several results relate to this problem. The closest to our model is the Precedence Constrained Scheduling problem introduced by Wikum [75], which studies a special case of MS-HPC-IO.

Theorem 1 ([75, Proposition 2.3])**.** *MS-HPC-IO is NP-complete. Even in the simplest case when $n_1 = 2$, and for all jobs J_i, $i \neq 1$, $n_i = 1$.*

II.B.2 Polynomial algorithms

In this section we present some instances where one can compute the optimal solution in polynomial time. We focus here on instances that are important for the HPC-IO problem. Several other specific instances have been studied by Wikum [75].

Case when $\forall i, n_i = 1$ When for all jobs J_i, $n_i = 1$, it is easy to see that any greedy solution that schedules the I/O as soon as they are available is optimal for MS-HPC-IO [75, Proposition 2.1].

Uniform jobs We study the case of uniform jobs which is a specific case of periodic jobs. Specifically we consider that there exists A, B s.t., for all i, j, $A_{i,j} = A$ and $B_{i,j} = B$. We can then write: $J_i = ((A, B)^{n_i})$. Those jobs can be used to represent some new types of workloads such as hyperparametrization in Machine Learning (see Section II.D.3 for more details). In this context, all jobs are part of a bigger job and are released at the same time. Because they are part of a bigger job, we are interested in solving MS-HPC-IO. In this section, w.l.o.g we assume that the jobs $(J_i)_{1 \leq i \leq m}$ are sorted by decreasing value of n_i.

Definition 3 (UNIFORM)**.** Given a set of jobs $(J_i)_{1 \leq i \leq m}$ s.t. $\forall i, r_i = 0, n_i \geq n_{i+1}$ and there exists A, B s.t., for all i, j, $A_{i,j} = A$ and $B_{i,j} = B$, UNIFORM is the problem of solving MS-HPC-IO.

Theorem 2. UNIFORM *can be solved in polynomial time.*

To show this result, we show that Algorithm 1 (HIERARCHICAL ROUND-ROBIN) solves the problem in polynomial time (Theorem 2.). The idea of HIERARCHICAL ROUND-ROBIN is to structure the schedule around the job with the largest n_i.

We start by scheduling each B operation of J_1. Then, we schedule before each of those B operations all B operations of jobs such that $n_i = n_1$. Finally, we schedule all remaining jobs in a round-robin fashion between $B_{1,1}$ and B_{1,n_1}. We present in Figure II.3 an example of such a schedule.

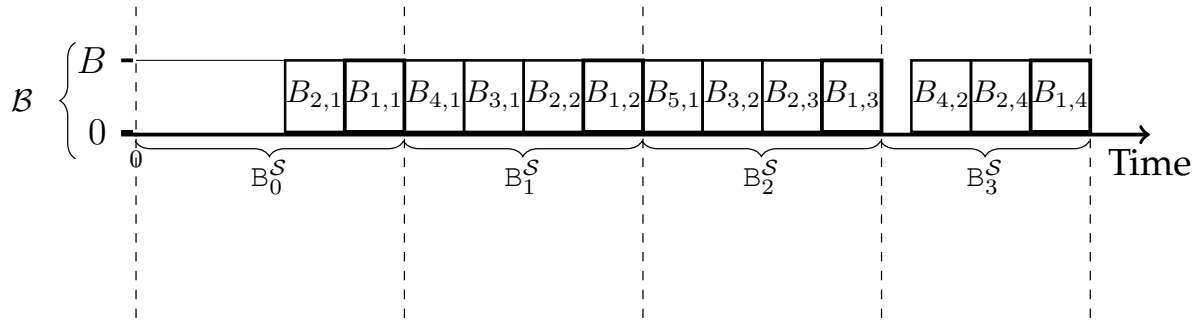


Figure II.3: Example of HIERARCHICAL ROUND-ROBIN schedule
$J_1 = (2.5, 1)^4$, $J_2 = (2.5, 1)^4$, $J_3 = (2.5, 1)^2$, $J_4 = (2.5, 1)^2$, $J_5 = (2.5, 1)^1$

In this section, we focus on showing formally this result. To do so:

1. We define of a cost function $\mathcal{C}$ (Def. 5) such that for all schedule $\mathcal{S}$, $C^{\mathcal{S}}_{\max} \geqslant \mathcal{C}(\mathcal{S})$ (Prop. 1);
2. We show that there exists an optimal schedule $\mathcal{S}_{opt}$ such that $\mathcal{C}(\mathcal{S}_{opt}) \geqslant \mathcal{C}(\mathcal{S}_{HRr})$ (where $\mathcal{S}_{HRr}$ is the schedule returned by HIERARCHICAL ROUND-ROBIN);
3. Finally, we show that $C^{\mathcal{S}_{HRr}}_{\max} = \mathcal{C}(\mathcal{S}_{HRr})$ (Prop. 2), showing the result.

Algorithm 1 HIERARCHICAL ROUND-ROBIN

procedure HRR($J_i = (\prod_{j \leq n_i} A_{i,j}, B_{i,j})$) ▷ $\forall i, j, n_i \geq n_{i+1}, A_{i,j} = A, B_{i,j} = B$
  Let $S_0, \cdots, S_{n_1-1}$ be n_1 empty stacks.
  $\mathrm{Id}_b \leftarrow 1$.
  for $i = 1$ to $|\{J_i\}|$ **do**
    if $n_i = n_1$ **then**
      for $j = 1$ to n_i **do**
        Add $B_{i,j}$ to S_{j-1}.
      end for
    else ▷ We do not schedule anymore on S_0.
      $\mathrm{Id}_e \leftarrow 1 + (\mathrm{Id}_b + n_i \mod (n_1 - 1))$ ▷ J_i is scheduled from S_{Id_b+1} to S_{Id_e}
      if $\mathrm{Id}_b \leq \mathrm{Id}_e$ **then**
        for $j = 1$ to n_i **do**
          Add $B_{i,j}$ to $S_{j+\mathrm{Id}_b}$.
        end for
      else
        for $j = 1$ to Id_e **do**
          Add $B_{i,j}$ to S_j.
        end for
        for $j = \mathrm{Id}_e + 1$ to n_i **do**
          Add $B_{i,j}$ to $S_{(n_1-1)-(n_i-j)}$.
        end for
      end if
    end if
    $\mathrm{Id}_b \leftarrow \mathrm{Id}_e$.
  end forreturn $\mathcal{S}_{HRr} = S_0 \cdot S_1 \cdot \cdots \cdot S_{n_1-1}$
end procedure

In the rest of this Section, we let $J_i = (0, \prod_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$ be a set of uniform jobs sorted by decreasing n_i. We denote by a (resp. b) the execution tasks of tasks $A_{i,j}$ (resp. $B_{i,j}$).

We introduce the notion of block:

Definition 4 (Block of a schedule $\mathcal{S}$ and its cost)**.** Given a schedule $\mathcal{S}$, for $k \in [[1, n_1]]$, we define the block $\mathtt{B}_k^{\mathcal{S}}$ to be:

- If $k = 1$, $\mathtt{B}_1^{\mathcal{S}}$ is the set of tasks scheduled to be executed before (including) $B_{1,1}$.
- Otherwise, $\mathtt{B}_k^{\mathcal{S}}$ is the set of tasks scheduled to be executed after (excluding) $B_{1,k-1}$ and before (including) $B_{1,k}$.

We define the cost of a block to be:

$$C(\mathtt{B}_k^{\mathcal{S}}) = \begin{cases} a + |\mathtt{B}_1^{\mathcal{S}}| & \text{if } k = 1 \\ max(a+b, |\mathtt{B}_k^{\mathcal{S}}| \cdot b) & \text{else} \end{cases}$$

We represent the notion of Blocks on Figure II.3

Definition 5 (Cost of a schedule)**.** Given a schedule $\mathcal{S}$, its cost is $\mathcal{C}(\mathcal{S}) = \Sigma_{k=1}^{n_1} C(\mathtt{B}_k^{\mathcal{S}})$, where C is the function cost of a block.

Proposition 1. *For any schedule $\mathcal{S}$, $C_{\max}^{\mathcal{S}} \geqslant \mathcal{C}(\mathcal{S})$.*

Proof. To obtain this result, one can observe that the blocks partition the schedule until B_{1,n_1}, and hence the total makespan is greater than the sum of the makespan of all blocks[1]. Then, we need to show that the makespan of each block is greater than the cost of each block, hence showing the result. This comes naturally from the fact the makespan of a block is necessarily greater than the maximum between (i) the total work that has to be performed during this block ($|\mathtt{B}_k^{\mathcal{S}}| \cdot b$), and (ii) the minimal length imposed by J_1 (an execution of $A_{i,j}$ and an execution of $B_{i,j}$). Hence, the makespan of a block is greater than its cost, showing the result. □

Definition 6 (Dominant schedules)**.** For UNIFORM, we say that a schedule is *dominant* if:

1. **Prop. (Dom.1)** The last task executed on platform $\mathcal{B}$ is B_{1,n_1};
2. **Prop. (Dom.2)** For all J_i, $(n_i - j + i) < n_1$, implies $B_{i,j}$ is executed after $B_{1,1}$.
3. **Prop. (Dom.3)** For all J_i such that $n_i = n_1$, $B_{i,1}$ is executed before $B_{1,1}$.

In practice *Dominant Schedules* are schedules that finish by the last operation of J_1, and that start by all first operations of *long* jobs and then by $B_{1,1}$.

Lemma 1. *There exists a dominant schedule which is optimal.*

Proof. We show the result in three steps:

1. First, we show that there exists an optimal algorithm which ends by the execution of B_{1,n_1};
2. Amongst those optimal algorithms, we show that there exists at least one where for all J_i such that $(n_i - j + 1) < n_1$, implies $B_{i,j}$ is executed after $B_{1,1}$;
3. Finally, amongst those, we show that there exists at least one such that for all J_i such that $n_i = n_1$, $B_{i,1}$ is executed before $B_{1,1}$.

[1] Where the makespan of block $\mathtt{B}_k^{\mathcal{S}}$ (resp. $\mathtt{B}_1^{\mathcal{S}}$) is naturally defined as the time between the beginning of the execution of $B_{1,k}$ on platform $\mathcal{B}$ and the beginning of the execution of $B_{1,k+1}$ on platform $\mathcal{B}$.

There exists an optimal algorithm that satisfies Prop. (Dom.1) We show the result by contradiction. Assume there does not exist an optimal schedule which ends by the execution of B_{1,n_1}.

Let $\mathcal{S}$ be an optimal schedule for UNIFORM that minimizes the number of operations following B_{1,n_1}. Let $B_{i,k}$ be the operation directly subsequent to B_{1,n_1} in the schedule.

If $k = n_1$, then because all $A_{i,j}$ are identical, for $1 \leq j \leq k$, we can permute all $B_{i,j}$ operations with $B_{1,j}$ without increasing the makespan, and the number of operations after B_{1,n_1} decreased strictly, contradicting the minimality of $\mathcal{S}$.

Otherwise, necessarily $k < n_1$ (indeed, by definition, for all i, $n_i \leq n_1$). In this case, necessarily there exist two consecutive operations of J_1 such that there are no operations of J_i between them. Let us call B_{1,n_1-j_0-1} and B_{1,n_1-j_0} those last operations. Then, because all jobs are identical, for $0 \leq j \leq j_0$, we can permute all $B_{i,k-j}$ operations with B_{1,n_1-j} operations without increasing the total makespan. In this new schedule, the number of operations after B_{1,n_1} decreased strictly, hence contradicting the minimality of $\mathcal{S}$.

We denote by A^1_{OPT} the non-empty set of optimal schedules that satisfy Prop. (Dom.1).

There exists a schedule in A^1_{OPT} that satisfies Prop. (Dom.2) Similarly, we show the result by contradiction. Assume that for all schedules of A^1_{OPT}, none satisfy Prop. (Dom.2).

Let $\mathcal{S} \in A^1_{\text{OPT}}$ that minimizes the number of operations $B_{i,j}$ that satisfy (i) $B_{i,j}$ is scheduled before $B_{1,1}$; (ii) $n_i - jn_1 - 1$. Let B_{i,j_0} be the last of these operations before $B_{1,1}$ in $\mathcal{S}$.

Then, because $(n_i - j_0 + 1) < n_1$, necessarily there exists $k < n_1$ such that there are no operations of J_i between $B_{1,k}$ and $B_{1,k+1}$. Let us denote by k_0 the smallest of such k. Then, for $j \in \{1, \cdots, k_0\}$we can permute in $\mathcal{S}$ all operations $B_{1,j}$ and B_{i,j_0-1+j} without increasing the schedule length. Indeed, there is no new idle time between any pair of operations $B_{1,j}$ and $B_{1,j+1}$ for $j < k_0$ (because $a_{1,j} = a_{i,j_0-1+j} = a$, nor between B_{1,k_0} and B_{1,k_0+1} because B_{1,k_0} was advanced in time while B_{1,k_0+1} did not move. Similarly, there is no new idle time created in the schedule between B_{i,j_0-1+j} and B_{i,j_0+j}. B_{i,j_0+k_0} is scheduled after B_{1,k_0+1} while B_{i,j_0-1+k_0} is scheduled where B_{i,k_0} was scheduled, so the time difference between them is greater than a.

Finally, this did not impact either any other jobs because the number of jobs on $\mathcal{B}$ between two occurrences on any other jobs was kept the same.

We can conclude that this transformation did not increase the execution time. In addition, it did not change the schedule after B_{1,k_0+1} where $k_0 + 1 \leq n_1$, hence Prop. (Dom.1) is still respected in this new optimal schedule. There was, however, one fewer job before $B_{1,1}$, contradicting the minimality of $\mathcal{S}$.

We denote by A^2_{OPT} the non-empty set of optimal schedules that satisfy both Prop. (Dom.1) and Prop. (Dom.2).

There exists a schedule in $A^2_{\mathbf{OPT}}$ that satisfies Prop. (Dom.3) Similarly, we show the result by contradiction. Assume that for all schedules of A^2_{OPT}, none satisfy Prop. (Dom.3).

Let $\mathcal{S} \in A^1_{\text{OPT}}$ that minimizes the number of operations $B_{i,1}$ that satisfy (i) $B_{i,1}$ is scheduled after $B_{1,1}$; (ii) $n_i = n_1$. Let $B_{i_0,1}$ be the first of these operations after $B_{1,1}$ in $\mathcal{S}$.

By a reasoning very similar to the one used to prove the existence of the set A^2_{OPT}, one can show that $\mathcal{S}$ can be chosen such that $B_{i_0,1}$ is the operation directly subsequent to $B_{1,1}$.

Because $n_{i_0} = n_1$, and because $\mathcal{S}$ satisfies Prop. (Dom.1), there exists $j_0 \geq 1$ such that B_{i_0,j_0} and B_{i_0,j_0+1} are scheduled between B_{1,j_0} and B_{1,j_0+1}.

Thanks to the property that $\forall i, j, a_{i,j} = a$, we can then create a new schedule whose execution time is not greater than that of $\mathcal{S}$ by permuting for $1 \leq j \leq j_0$, $B_{i_0,j}$ and $B_{1,j}$. This schedule still satisfies Prop. (Dom.1) (we have not modified the location of B_{1,n_1}), and Prop. (Dom.2) (the only task that was moved before $B_{1,1}$ is $B_{i_0,1}$), contradicting the minimality of $\mathcal{S}$.

Finally, this concludes the proof that there exists an optimal schedule that is dominant.

□

Lemma 2. *Denote by $l_1 = |\{J_i | n_i = n_1\}|$ and $\mathcal{S}_{HRr}$ the solution returned by* HIERARCHICAL ROUND-ROBIN. *Let $r_1 = (\sum_i n_i - l_1) \mod (n_1 - 1)$, and $q_1 = \lfloor \frac{\sum_i n_i - l_1}{(n_1-1)} \rfloor$ Then, we have the following results:*

- $|B_1^{\mathcal{S}_{HRr}}| = l_1$,
- *for $j = 2$ to $r_1 + 1$,* $|B_j^{\mathcal{S}_{HRr}}| = q_1 + 1$,
- *for $j = r_1 + 2$ to n_1,* $|B_j^{\mathcal{S}_{HRr}}| = q_1$.

Proof. This is a direct consequence from Algorithm 1. One can notice that $\mathtt{B}_k^{\mathcal{S}_{HRr}}$ corresponds to S_{k-1} as returned at the end of the execution.

Hence, $\mathtt{B}_1^{\mathcal{S}_{HRr}}$ only contains the first operation of jobs of length n_1 (hence l_1 operations), and the rest of the blocks share the remaining operations minus those l_1 operations, hence the result. □

Lemma 3. *Given $\mathcal{S}$ a* dominant schedule, *then $\mathcal{C}(\mathcal{S}) \geq \mathcal{C}(\mathcal{S}_{HRr})$.*

Proof. In this proof we use the definition of l_1, q_1 and r_1 as defined in Lemma 2.

Let $\mathcal{S}$ be a dominant schedule. Denote by $p_{\min} = \min_{k=2}^{n_1}\{|\mathtt{B}_k^{\mathcal{S}}|\}$ (resp. $p_{\max} = \max_{k=2}^{n_1}\{|\mathtt{B}_k^{\mathcal{S}}|\}$, the smallest (resp. largest) block size for all blocks of $\mathcal{S}$ but the first one.

We show the result by recurrence on $|p_{\max} - p_{\min}|$. By definition of a dominant schedule, we know that $\sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}}| = \sum_i n_i - l_1$, hence necessarily $p_{\min} \leq q_1 \leq p_{\max}$.

By definition of q_1 and r_1, if $p_{\max} - p_{\min} \leq 1$, then $p_{\min} = q_1$ and there are exactly r_1 blocks of size $p_{\max}$ and $n_1 - r_1$ blocks of size $p_{\min}$. Hence, $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}_{HRr})$. In the following we assume that $p_{\max} - p_{\min} > 1$. In particular we have: $p_{\min} \leq q_1 < q_1 + 1 \leq p_{\max}$.

If $p_{\min} \cdot b \geq a + b$ **(resp.** $p_{\max} \cdot b \leq a + b$**)** Then, we have:

$$\sum_{k=2}^{n_1} C(\mathtt{B}_k^{\mathcal{S}}) = \sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}}| \cdot b = b \left((\sum_i n_i) - l_1 \right) = b \sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}_{HRr}}| = \sum_{k=2}^{n_1} \mathcal{C}(\mathtt{B}_k^{\mathcal{S}_{HRr}})$$

(resp. $\sum_{k=2}^{n_1} C(\mathtt{B}_k^{\mathcal{S}}) = \sum_{k=2}^{n_1} a + b = \sum_{k=2}^{n_1} \mathcal{C}(\mathtt{B}_k^{\mathcal{S}_{HRr}})$), meaning that $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}_{HRr})$.

Else, $p_{\min} \cdot b < a + b < p_{\max} \cdot b$ In this case, because $|p_{\max} - p_{\min}| \geq 2$, we can show that the cost is strictly greater to the cost of a solution with one element fewer in the largest block, and one more element in the smallest block. This can be done recursively until one of the initialization case as seen above (either $|p_{\max} - p_{\min}| \leq 1$, $p_{\min} \cdot b \geq a + b$, or $p_{\max} \cdot b \leq a + b$) for which we have shown that the cost is equal to $\mathcal{C}(\mathcal{S}_{HRr})$.

Indeed, assume the cost of the smallest block increases by $0 \leq \delta < b$ (resp. cost of the largest bock decreased by $0 < \delta \leq b$). Then, $a+b \leq (p_{\max}-1) \cdot b$ (resp. $(p_{\min}+1) \cdot b \leq a+b$), and the cost of the largest block decreased by b (resp. the cost of the smallest block did not increase). Hence, the total cost decreased by $b - \delta > 0$ (decreased by $\delta > 0$).

Again, the path of solutions may not theoretically exist, however this process shows that their cost is indeed greater than that of $\mathcal{S}_{HRr}$. □

Proposition 2. $\mathcal{C}(\mathcal{S}_{HRr}) = C_{\max}^{\mathcal{S}_{HRr}}$

Proof. We study the stacks $S_0, \cdots, S_{n_1-1}$ as returned by Algorithm 1. Note that we have seen that there execution time is necessarily at least equal to their cost because of J_1. We now show that this time is enough for a successful execution of the schedule.

The time to execute S_0 is exactly $\mathcal{C}(S_0)$, indeed all jobs in this stack are executed for the first time, hence we need to wait for a time a, then all I/O operations are ready and we can execute them consecutively (taking a time $|S_0| \cdot b$.

We then show the result on the other stacks by studying the l^{th} element from the bottom of the stack (the first element of each stack S_k is $B_{1,k+1}$).

Given stack S_k, denote by $B_{i,j}$ its l^{th} element:

- Either $j = 1$, in which case it was ready since S_0 and there are no additional time constraints;
- Or $B_{i,j-1}$ was put on stack S_{k-1}, then it was at the l^{th} position of the stack because the stack is balanced. In which case, there are exactly $l-1$ (resp. $|S_k|-l$) operations on stack S_{k-1} (resp. S_k) between those two operations, hence a total time of $(|S_k| - 1) \cdot b$. Hence, we need an idle time at the beginning of the execution of S_k of length $\max(0, a - (|S_k| - 1) \cdot b)$, and an execution time for S_k of $\mathcal{C}(S_k)$ is enough for its successful execution.
- Finally, with the round robin property, $B_{i,j-1}$ could be scheduled on stack $S_{k'}$ where $k' < k - 1$. In this case the time constraint is also respected because S_{k-1} takes by definition more than a units of time.

Hence, the result, we have shown that an execution time equal to the cost for each task was enough to satisfy all the time constraints.

□

Proof of Theorem 2. There exists an optimal schedule $\mathcal{S}_{opt}$ to UNIFORM, such that (i) $C_{\max}^{\mathcal{S}_{opt}} \geq \mathcal{C}(\mathcal{S}_{opt})$ (Prop. 1); (ii) $\mathcal{C}(\mathcal{S}_{opt}) \geq \mathcal{C}(\mathcal{S}_{HRr})$ (Lemma 1 and Lemma 3). Finally, we have seen (Prop. 2) that $\mathcal{C}(\mathcal{S}_{HRr}) = C_{\max}^{\mathcal{S}_{HRr}}$, proving that HIERARCHICAL ROUND-ROBIN is optimal. □

II.C Approximation algorithms for MS-HPC-IO

We have seen in Section II.B that MS-HPC-IO was in general intractable. A natural question to this is whether there exist efficient approximation algorithms. In this part, we show some results on list-scheduling algorithms, then discuss a specific framework of algorithms, periodic algorithms.

Definition 7 (Approximation algorithm)**.** For a maximization (resp. minimization) problem $\mathcal{P}$, we say that an algorithm $\mathcal{A}$ is a λ-approximation algorithm, if for any instance $I \in \mathcal{P}$, $\mathcal{A}(I) \leq \lambda \mathcal{A}_{\text{OPT}}(I)$ (resp. $\mathcal{A}(I) \geq \lambda \mathcal{A}_{\text{OPT}}(I)$) (where $\mathcal{A}_{\text{OPT}}$ is an optimal algorithm for $\mathcal{P}$).

II.C.1 List Scheduling algorithms

We start by considering *list scheduling* strategies (also called *greedy*) which are often considered the most natural algorithms: at all time, either the machine $\mathcal{B}$ is busy or no work of type $\mathcal{B}$ is available. When the machine becomes idle and some multiple operations are available, the machine sorts them (and schedule them) following a priority order.

Theorem 3. *Any list-scheduling algorithm is a 2-approximation for MS-*HPC-IO *and this ratio is tight.*

Proof. First, we show that any list-scheduling algorithm is at best a factor two of the optimal for MS-HPC-IO.

We create the instance I_ε: $J_1 = ((A_{1,1} = 0, B_{1,1} = 1))$, $J_2 = ((A_{2,1} = \varepsilon, B_{2,1} = \varepsilon) \cdot (A_{2,2} = 1, B_{2,2} = 0))$. The makespan of any list-scheduling algorithm is: $2 + \varepsilon$. Indeed, at $t = 0$, a list-scheduling algorithm has to schedule $B_{1,1}$ because it is the only operation ready. Then, once it is done, it can schedule $B_{2,1}$, which will be followed by the execution of $A_{2,2}$ and $B_{2,2}$.

On the other hand, an optimal schedule waits for ε units of time so it can schedule $B_{2,1}$ first. Then, it schedules $B_{1,1}$ while $A_{2,1}$ executes. The total execution time is $1 + 2\varepsilon$. Hence, the approximation ratio is at least:

$$\lambda = \sup_{\varepsilon > 0} \frac{2 + \varepsilon}{1 + 2\varepsilon} = 2$$

We now show that any list-heuristic algorithm is at most a 2-approximation. Given an instance of the problem, let $C_{\max}^{List}$ be the makespan of a list-scheduling algorithm and $C_{\max}^{\text{OPT}}$ be the makespan of an optimal algorithm.

Necessarily, $C_{\max}^{\text{OPT}} \geq \max_i(\sum_j a_{i,j} + b_{i,j})$ which is the minimal time needed for the longest application J_i. We focus on the occupation of platform $\mathcal{B}$. $C_{\max}^{List} = \sum_i \sum_j b_{i,j} + t_{\text{idle}}$, where t_{idle} is the time where platform $\mathcal{B}$ is waiting for work. Let $B_{i_0,n_{i_0}}$ be the last operation scheduled on $\mathcal{B}$. Then, necessarily, $t_{\text{idle}} \leq \sum_{j=1}^{n_{i_0}} a_{i,j}$.

Hence, we have:

$$C_{\max}^{List} = \sum_i \sum_j b_{i,j} + t_{\text{idle}} \leq C_{\max}^{\text{OPT}} + \sum_{j=1}^{n_{i_0}} a_{i,j} \leq 2C_{\max}^{\text{OPT}}$$

□

II.C.2 Periodic algorithms

In this part, we focus on periodic applications as defined in Section II.A. These applications are very frequent in our target framework, High-Performance Computing (the most common example is that of applications that store their checkpoint at regular intervals for resilience purpose [19]). To tackle them, we study a specific sort of algorithms: *Periodic algorithms*. Indeed it has been shown that those algorithms have many efficient property such as a low memory and compute overhead with excellent performance when the number of operations per jobs is very high [9]. We are interested here in giving some theoretical results that motivates these recent results.

We start by showing that in some context, those algorithms are efficient approximations for the MS-HPC-IO problem.

We define formally a periodic algorithm:

Definition 8 (Periodic Algorithm)**.** Given a periodic instance $J = ((a_i, b_i)^{k_i \cdot n})$.

A periodic algorithm $\mathcal{P}$ constructs a *period* which is a schedule of $J = ((a_i, b_i)^{k_i})$: then returns a schedule built by n periodic repetition of the period.

Periodic algorithms for MS-HPC-IO We start by considering periodic jobs whose n_i are all equal. In this case HIERARCHICAL ROUND-ROBIN is a periodic algorithm.

Theorem 4. HIERARCHICAL ROUND-ROBIN *is a* $1 + 1/n$*-approximation algorithm for MS-*HPC-IO *where all jobs are periodic with the same number of periods (there exists* n*, such that* $\forall i,\ J_i = ((A_i, B_i)^n)$*), and the bound is tight.*

Proof. First, we discuss the way of ordering tasks within a period and then discuss the performance of such scheduling algorithms.

- In the following, we call "idle time" of a schedule $\mathcal{S}$, the time $t_i(\mathcal{S}) = MS_{\mathcal{S}} - \sum_i nb_i$

- In PERIODIC, all jobs have only one task in each period. We can define the order $\prec$: $i \prec j$ if and only if b_i appears before b_j in the period.

The overall idle time of the periodic schedule is:
$t_i(Periodic) = (n-1).\underset{i}{max}\left(a_i - \underset{j\neq i}{\sum} b_j\right) + \underset{k}{max}\left(a_k - \underset{k\prec j}{\sum} b_j\right)$

The order within a period does not change the overall idle time, therefore we can sort tasks by non-increasing A-task length in a period with gives:

$$t_i(Periodic) \leq (n-1).\underset{i}{max}\left(a_i - \underset{j\neq i}{\sum} b_j\right) + \underset{k}{max}\left(a_k\right)$$

Given an optimal schedule $\mathcal{S}_{opt}$, the idle time is:
$t_i(\mathcal{S}_{opt})\underset{i}{max}\left(na_i - n\underset{j\neq i}{\sum} b_j + \underset{j\prec i}{\sum} b_j\right) \geq n.\left(\underset{k}{max}\left(a_k - \underset{j\neq k}{\sum} b_j\right)\right)$

where i is the last task running on A and i_1 is its first iteration. Therefore, using straightforward bounds, the difference between these periodic and opt is at most:
$n.\underset{i}{max}\left(a_i\right) - n\left(\underset{k}{max}\left(a_k - \underset{j\neq k}{\sum} b_j\right)\right) \leq n.\underset{i}{max}\left(a_i\right)$ The optimal makespan is at least $n.\underset{i}{max}\left(a_i + b_i\right)$

□

Remark 1. *One can notice that* HIERARCHICAL ROUND-ROBIN *is asymptotically optimal for MS-*HPC-IO *when all jobs are periodic with the same number of periods. In addition, one can slightly improve the result by sorting the jobs by decreasing values of* a_i.

II.D Experimental evaluation

In this section we present the experimental evaluation of the proposed solutions. To evaluate them we have designed a simulator that implements the model described in section II.A i.e a virtual platform with two machine types, with no competition on resource $\mathcal{A}$ and competitive, exclusive accesses on machine $\mathcal{B}$.

II.D.1 Heuristics

For the purpose of evaluation, we compared several heuristics, list-scheduling heuristic as well as very simple periodic algorithms. These heuristics use different *priority order* to choose which task to execute next. One of them is Johnson's priority order

Definition 9 (Johnson's order)**.** Given a set of couples (a_i, b_i), divide the values into two disjoint groups G_1 and G_2 , where G_1 contains all couples (a_i, b_i) with $a_i \leqslant b_i$, and G_2 contains all couples (a_j, b_j) with $a_j > b_j$. Order the couples in a sequence such that

the first part consists of the values in G_1 , sorted in nondecreasing order of a_i, and the second part consists of the values in G_2 , sorted in non-increasing order of b_j.

The reason why Johnson's order is considered is because if jobs are $J_i = (a_i, b_i)^1$, it is known that the schedule using Johnson order minimizes the completion time of the flowshop. [76].

List scheduling In list scheduling policy, as soon as I/O is free, we execute the most critical, available application. We used different orders to define the criticality of a given application:

- **FIFO**: the applications I/O are executed in the order of their request.
- **Johnson**: the application I/Os are executed following Johnson's order (see definition 9)
- **Most Remain**: When scheduling an I/O, pick in priority the application with the most remaining work to do.

Periodic We also use a simple variation of periodic heuristics defined in II.C.2. Given an instance $J_i = (a_i, b_i)^{n_i}$, each period of the periodic algorithm contains exactly one task for each job until one of the job is completed. The jobs are sorted following the three orders used for list-scheduling heuristics (FIFO, Johnson, Most Remain). The completion of a job or the release of a new one does not change the relative order of the other. Hence, the period holds after such events (with the exception of the completed job who is not running anymore).

Best effort With the best effort strategy, there is no schedule of I/O accesses. Instead of waiting their turn to perform I/O operations, concurrent applications accessing the storage system share equally the bandwidth without additional loss. If k applications are performing I/O operations, an application with b amount of I/O will have, after t units of time, $b - \frac{t}{k}$ remaining amount of I/O. The best effort strategy models what happens in real systems when there is no congestion control or I/O scheduling at the level of the applications.

II.D.2 Scenarios/Use-case and instantiation

Applications are modeled by their computation amount, I/O durations, and their number of periods. An input file describes an *instance* of the problem as a set of m applications and is generated according to table II.1. We have two different cases that represent realistic settings.

The UNIFORM case is used for a machine learning multi-parameter training and covers the results of Section II.B.2.

cases	m	a_i	b_i	n_i	r_i	#instances
General	$\mathcal{U}$(2,15)	$\mathcal{U}$(1,20)	$\mathcal{U}$(0.1,1)a_i	$\mathcal{U}$(5,150)	0	1000
UNIFORM	$\mathcal{U}$(2,15)	$\mathcal{U}$(1,20)	$\mathcal{U}$(0.1,1)a_i	$\mathcal{U}$(100,200)	0	1000

Table II.1: Parameters used for input generation ($u(a, b)$ stands for drawing uniformly in $[a, b]$)

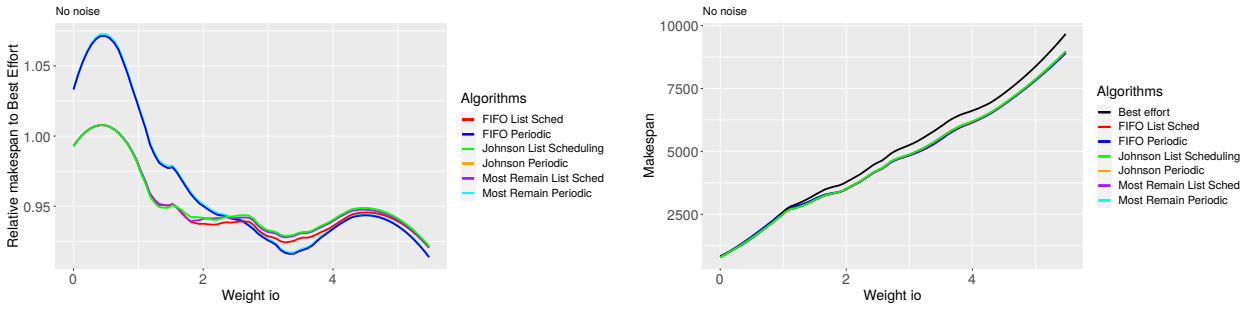
II.D.3 Results

In Figure II.4, we present the makespan for the general case. The presented graph is the smoothed conditional means on a set of 1000 instances of each case as a function of the weight of I/O, W ,that accounts for a normalize way of measuring the amount of I/O:

$$W = \sum_i \frac{\sum_j b_{i,j}}{\sum_j a_{i,j} + b_{i,j}}$$

In this figure, we see that, when the weight of I/O is small, the best effort strategy provides the fastest makespan. This is due to the fact that when there are few I/O, scheduling them is not very useful. However, as soon as the amount if I/O increases, the scheduling strategies improves and outperform the best effort one. Moreover, we see two groups of curves. Periodic schedules and list-scheduling ones. The periodic strategies, *FIFO Periodic*, *Johnson Periodic* and *Most Remain Periodic* are superposed. If we compare these two sets of strategies, we see that when the amount of I/O is small relative to the total of work, list scheduling perform better than periodic strategies and when the weight of I/O increases the periodic strategies are better than the list-scheduling ones. Indeed, when there is few I/O the periodic schedule can force an application to wait for their turn while when there is a high amount of I/O, the short view of the problem by list scheduling algorithm hinder their capacity to handle I/O burst. Whereas all strategies have overall the same makespan evolution, the normalization with best effort (Figure II.4a) accentuate a variation in performance around W=3.This can be explained as follows: after a certain threshold in I/O weight, bandwidth is always busy, therefore all applications have to wait for I/O accesses which mitigate a little the gain. Whereas for low weights, our algorithms reduce the I/O contention, and for high I/O weights it provides a better repartition on I/O operations compared to the best effort strategy. If we look at the absolute tendencies (see Figure II.4b), we observe that the difference bewteen best-effort and the other strategies is globally increasing with the weight of I/O. Some fluctutations for the relative charts are also due to the impact of the input workload.

Remark 2. *In this part, we focus on the scheduling of I/O operations. Therefore the way of measuring the I/O weight of applications takes only applications phases into account and is agnostic to the platform state. In future chapters, as we will introduce platform constraints to our reflection, we will prefer a refined definition (see Equation (III.5)).*



(a) Relative performance to the Best effort strategy

(b) Absolute performance

Figure II.4: Policies performance comparison on generic inputs for the makespan

Uncertainty and noise In our implementation, list scheduling and periodic policies assume that the I/O and computation duration are known in advance. However, in practice these values can never be known with a complete certainty [78]. To model this uncertainty we have added noise to I/O and computation duration. This means that each computation or the I/O phase can be subject to a variation in the range of the noise value around the expected, periodical amount. This variation is determined independently for each phase. It is generated based on a seed that is included with the application specification in order to be reproducible. Indeed, we want this variation to be the same without any concern of the application order. It is unclear what is the right modeling for this variability. In the literature, some authors [78] propose a statistical modeling via a large array of distributions (normal, gamma, lognormal etc). Here we choose to model them with uniform distribution: we expect such noise model to be worse for our policies than a distribution with concentrated density around the mean and hence would lead to an lower bound in terms of performance.

In Figure II.5, we present the results with respectively 20% and 50% of noise using the same inputs as for the one in Figure II.4.

We see that adding noise slightly degrades the performance when the amount of I/O is small compared to the total amount of work. However, when the weight of I/O increases we observe relatively similar performance compared to the case without noise. This means that our strategies are robust to the uncertainty of the duration especially when the amount of I/O is large.

Machine Learning Use-Case We describe here a use case where a set of applications is launched at the same time and perform periodic I/O. The goal is to train, in parallel, several deep-learning networks (DLNs) on the same dataset. It works as follows. A set of m nodes of a parallel machine is reserved. m DLNs are generated and trained separately on each node. The goal is to find the best network among the m ones. Therefore, they are trained on the same dataset. Each DLN access a subpart of the dataset from

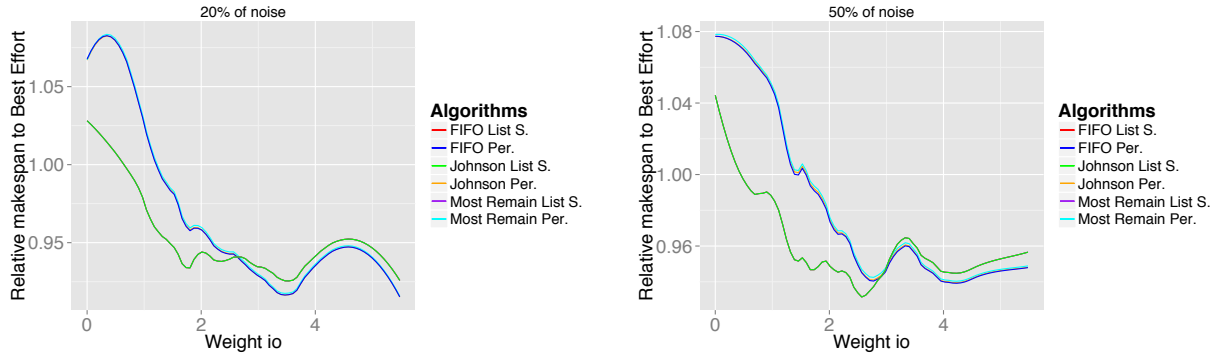


Figure II.5: Policies performance comparison on generic inputs for the makespan relative to the Best effort strategy with uniform noise on the computation or I/O duration

the storage and train itself on this subpart using supervised learning (e.g. with a gradient descent). Then, if the network has not converged it fetches another subpart of the dataset and iterate the learning part. As, for a given DLN, the subpart is of the same size, the IO time (without congestion) and learning time is constant across iterations. However, as each DLN is different (e.g. in terms of topology and meta parameters) the number of iterations is different across DLN. Therefore, according to our nomenclature this use-case fits the UNIFORM case: $J_i = (A, B)^{n_i}, i \in [1, m]$.

In Figure II.6, we compare best effort and the FIFO list scheduling strategies which are both non-clairvoyant (they do not know in advance the number of periods) against the HIERARCHICAL ROUND-ROBIN for which the closed form of the makespan is given as follows. We are in the UNIFORM case: the set of jobs is $J_i = (0, (a, b)^{n_i}$. We denote by $n = \max_i n_i$, $l = |\{J_{i_0} \, | n_{i_0} = n\}|$ (the number of jobs of maximum n_i), $q = \frac{(\sum_i n_i) - l}{n-1}$ and $r = ((\sum_i n_i) - l) \mod (n-1)$. Then, the makespan of HIERARCHICAL ROUND-ROBIN $C_{\max}^{HRr}$ is:

$$C_{\max}^{HRr} = a + l \cdot b + (n - 1 - r) \cdot \max(a + b, qb) + r \cdot \max(a + b, (q + 1)b)$$

According to Theorem 2, HIERARCHICAL ROUND-ROBIN is asymptotically optimal. Moreover, the FIFO list-scheduling is a 2-approximation algorithm (Theorem 3). For this use case, we see that despite the fact that the FIFO list-scheduling is non-clairvoyant it provides a makespan very close to HIERARCHICAL ROUND-ROBIN (less than 10% slower). Concerning the best effort strategy, we see that it performs worse than FIFO list-scheduling and up to 60% slower than HIERARCHICAL ROUND-ROBIN. Indeed, in this case, the access of the I/O is synchronized and the best-effort strategy maintain this synchronization and hence the I/O contention during the whole execution of the instance.

To test the case where we can have desynchronization due to uncertainty in computation or I/O execution, we have added 20% of uniform noise on these two costs. The results are presented on Figure II.6. In this case, we see that the noise has almost no im-

pact on the FIFO list-scheduling strategy. For the best effort strategy, we see that it has a better performance than without noise but it is still worse than the FIFO list-scheduling. This shows that the best effort strategy does not behave well in case of high congestion of the network.

II.E Summary and perspectives

In this chapter, we have studied the problem of scheduling I/O access for applications that alternate computation and I/O. We have formally described the problem as scheduling bi-colored chains. Then, we have studied theoretical results. Despite the fact that the general case is NP-complete, we have provided an optimal algorithm for the UNIFORM case. Moreover, we have studied two classes of strategies: periodic and list scheduling ones. We have shown that any list-scheduling algorithm is a 2-approximation and that HIERARCHICAL ROUND-ROBIN is asymptotically optimal for the periodic case. We have also studied different order for instantiating several heuristics (both periodic and list-scheduling ones).

We have experimentally tested, through simulations, the proposed approaches on realistic cases. We have shown that periodic approaches are the best ones when the relative amount of I/O is high and that the best effort strategy is the worst one. Moreover, we have studied the case where the input is not known with complete certainty but subject to noise. In this case the proposed approaches are shown to be robust. Last, we have studied the case of a distributed learning phase for deep-learning. Results show that the FIFO list-scheduling strategy is very close to the optimal one (despite being non-clairvoyant) and much better than the best effort.

The most important contribution of this chapter is the introduction of a model for HPC applications based on the simple observation of their two phase behavior. Convincing results have been proposed for periodical applications.

We sketched tracks to deal with non-periodic applications as well but it makes the problem significantly more complex, not theoretically but in practice, to design solutions. However, we showed that this model stays valid when there is some noise tampering with the periodicity. This chapter serves as a preamble for the following. The model will be re-used, adapted and enriched throughout this manuscript. The next step, described in Chapter III, is to incorporate machine consideration in our models. Taking computational resource constraints into account, we address the problem of I/O-aware application mapping onto the platform in order to tackle the contention upstream.

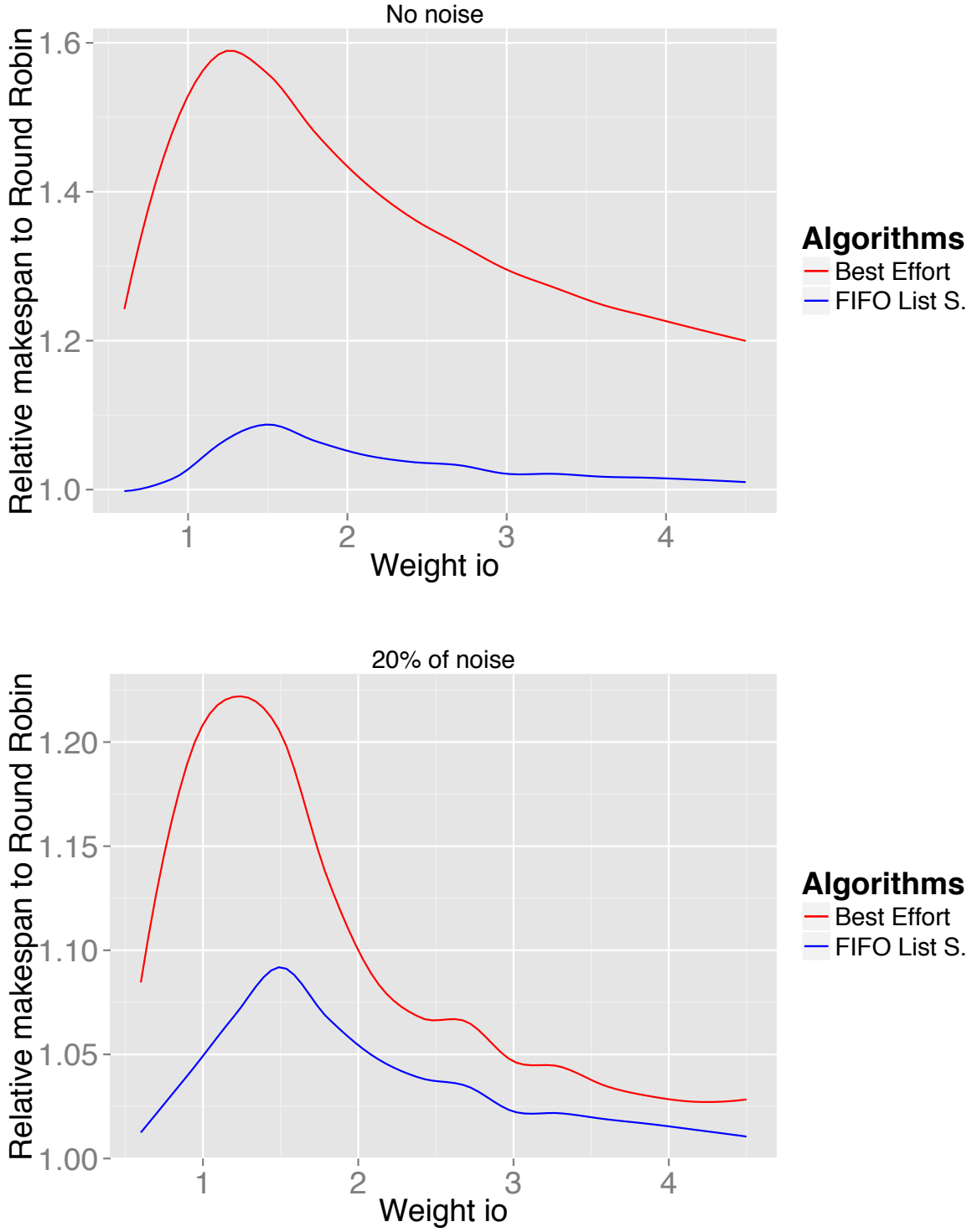


Figure II.6: Policies performance comparison of the ML use case for the makespan relative to HIERARCHICAL ROUND-ROBIN (top no noise, bottom 20% of uniform noise).

Chapter III

Mapping with pack scheduling - Resource control

In the previous chapter, we studied scheduling algorithms to avoid I/O contention. Indeed, I/O accesses are exclusive and as long as one operation is using the I/O node, additional requests get delayed. However, we assume that application were mapped beforehand. In this chapter, we include the application mapping in our study in order to prevent contention upstream. However, it makes sense to discuss strategies only if the application behavior is stable. That is to say, applications relative performance must not depend on minor parameter changes. To this extent, we resume previous work in order to define under what condition to perform experiments.

In the end, we propose a new solution aimed at reducing I/O interference via globally coordinated I/O access operation by studying two problems. The mapping problem consists in selecting the set of applications that are in competition to access sequentially the I/O nodes. The scheduling problem consists then, given I/O requests on the same resource, in determining the order of these accesses to minimize the I/O time. The main contributions of this chapter are a mathematical model and a packing algorithm to optimize the mapping of applications compute nodes to I/O nodes as well as a thorough experimental study of several I/O scheduling policies to order sequences of I/O operations that must be executed through each I/O node. Then we designed and validated a simulator to perform the evaluation at a larger scale. We then used this simulator to perform additional evaluations of the impact of the mapping strategy. The evaluation results with a single I/O node, show that our cross-layer approach greatly reduces the stretch of the machine while slightly degrading the makespan compared to the standard First-Fit algorithm. Meanwhile, with several partitions (and I/O nodes), our bandwidth-aware strategy performs better for both metrics.

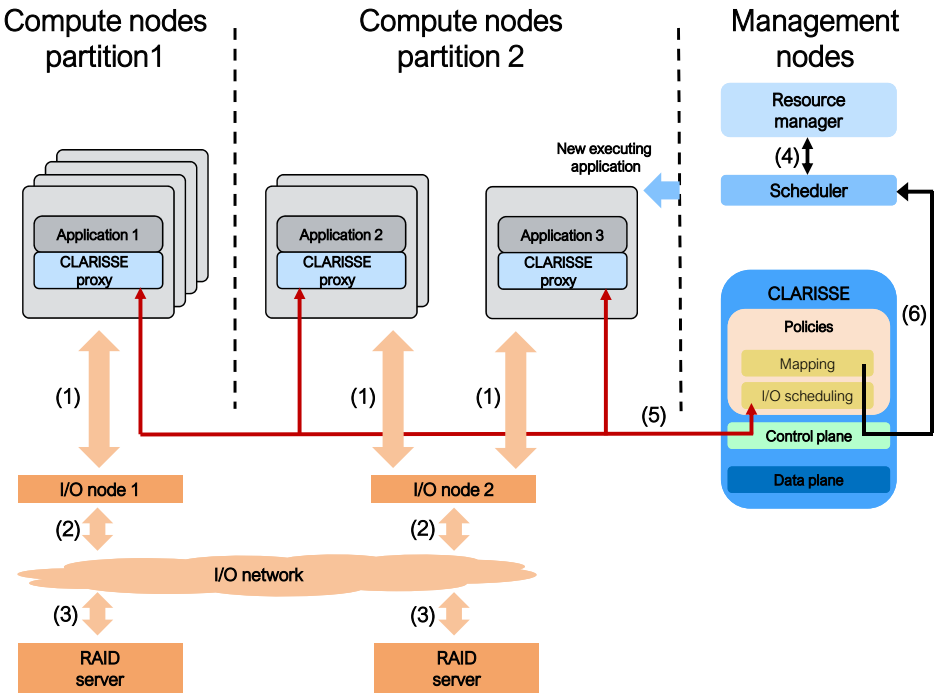


Figure III.1: Framework overview.

III.A Architecture overview

Figure III.1 shows the architecture of the proposed framework. The resources if the considered parallel machine are divided into management nodes, that execute the software management tools, and compute nodes, that execute the applications. In this figure, three running applications with 4, 2 and 1 processes are illustrated. Using a similar scheme than in large HPC infrastructures, the compute nodes are divided into partitions (two in the figure). Each partition is associated to one I/O node that is responsible for managing the application I/O accesses (1) and translating them into requests to the storage servers (2) that are subsequently sent to the disks (3).

On the management side, the administration tools include a resource manager for monitoring the system and allocating compute nodes for new applications. The resource manager communicates with the scheduler (4), responsible for determining what is the next application to be executed and on which resources. The third component is controlling the I/O accesses to the bandwidth, effectively coordinating the I/O flow and enforcing I/O scheduling policies. In addition, the control plane includes a communication line with the scheduler (arrow 6). The policy layer includes mapping techniques that guide the scheduler to allocate the new executing application in certain partitions, with the aim of reducing the number of conflicts accessing the I/O node.

This model can be implemented with a middleware. The arcos team [1] based in University Carlos III of Madrid implemented CLARISSE [39], a middleware for enhancing I/O flow coordination and control in the HPC systems. For this study, we collaborated with them, implementing I/O scheduling policies in the adequate CLARISSE layer and using their code as a framework for experimental study.

CLARISSE decouples the policy, control, and data layers of the I/O stack software in order to simplify the task of globally coordinating the parallel I/O on large-scale HPC platforms. At the time we did this study, it incorporated a public-subscribe protocol in the control plane for coordinating the I/O, and several I/O scheduling policies in the policy layer. Each running application included a CLARISSE proxy that wrapped the application I/O calls and communicates with CLARISSE via the control plane (arrow 5). By means of this channel, the I/O scheduling policies can impose a multi-criteria I/O access order to the running application based on different performance metrics.

As an illustrative example, let's assume that applications 1 and 2 are initially running in the system, and application 1 is more I/O intensive than application 2. Then, application 3, that is also I/O intensive, is ready to be executed. In order to balance the accesses to both I/O nodes, the mapping policy determines that is better to place application 3 in partition 2, avoiding risk of contention in the I/O node related to application 1. In a second step, when all the applications are being executed, application 2 and 3 compete for the I/O resources in the same partition. Note that conflicts in the access to the second I/O node may arise. Using the new CLARISSE's I/O scheduling policy introduced in this paper, the I/O accesses of both applications are coordinated with the aim of reducing the number of conflicts.

III.B Formal definition

In this section we present a mathematical model of the problem considered. The machine model behavior has been verified experimentally to be consistent with the behavior of Intrepid and Mira, supercomputers at Argonne [29], as well as with a supercomputer at Mellanox [9].

III.B.1 Machine Model

We consider a parallel platform structured as follows: R I/O nodes $(n_1^{\text{io}}, \ldots, n_R^{\text{io}})$ are available to perform I/O operations from the compute nodes to the parallel file system. Given $j \in \{1, \ldots, R\}$, each I/O node n_j^{io} has a bandwidth b_j for these operations, which is shared among P_j compute nodes (for a total of $\sum_{j=1}^{R} P_j$ compute nodes).

In this work we assume that the I/O bandwidth is homogeneous, that is, $\forall j, b_j = b$, as well as the number of compute nodes associated to each I/O node ($\forall j, P_j = P$). We represent this architecture model on Figure III.2.

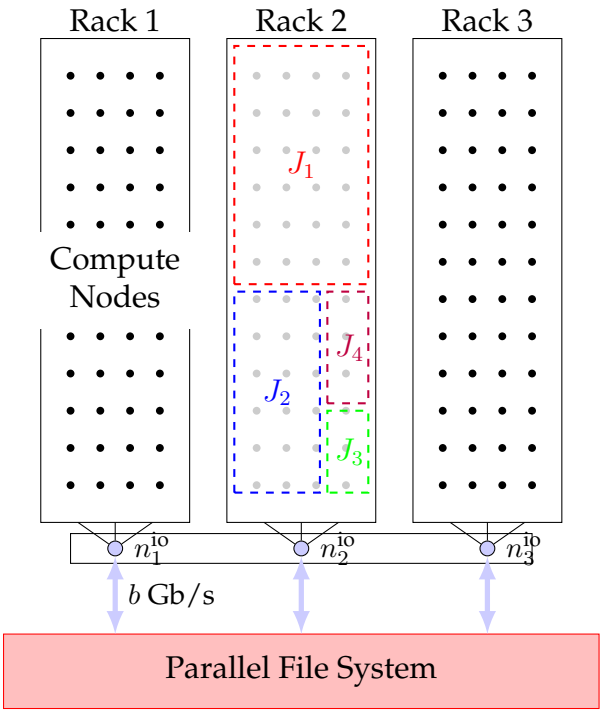


Figure III.2: Schematic of the architecture. Jobs J_1, J_2, J_3 and J_4 compete for the bandwidth available on n_2^{io}.

III.B.2 Applications

We consider a batch of scientific applications that need to run simultaneously onto the parallel platform. As in the previous chapter, applications consists of a series of consecutive *non-overlapping* phases: (i) a compute phase (executed on the compute nodes); (ii) an I/O phase (a transfer of a certain volume of I/O using the available I/O bandwidth) which can be either reads or writes.

Formally, we have a set of n jobs $\{J_1, \ldots, J_n\}$. Each job J_i requests Q_i compute nodes for its execution. J_i consists of n_i successive, blocking and non-overlapping operations: (i) $A_{i,j}$ (a compute operation that lasts for a time $a_{i,j}$); $B_{i,j}$ (an I/O operation that consists in transferring a volume $b_{i,j}$ of data). Therefore, if the bandwidth available to J_i to transfer its I/O to the PFS is equal to b, the time T_i needed for the total execution of J_i is:

$$T_i(b) = \sum_{j \leq n_i} a_{i,j} + \frac{b_{i,j}}{b}. \tag{III.1}$$

However, in general the I/O bandwidth is shared amongst several applications and it may incur delays to the execution of J_i.

III.B.3 Optimization problem

In this work, we consider that each job must be scheduled on the compute nodes associated to a single I/O node (contiguity), but that the I/O nodes can be shared amongst several applications. In addition, following the motivational example on I/O interference presented in section I.A.4, we do not allow simultaneous bandwidth sharing amongst applications (i.e. on a given I/O node, only one application is performing I/O at the same time), and we do not allow preemption of I/O (once an application has started to perform I/O, it has to finish its transfer).

A *schedule* is the solution of two allocation problems:

- The *mapping* problem, which consists in choosing for each application the allocation of compute nodes (as depicted in Figure III.2);
- The *scheduling* problem, which, for a given I/O node, consists in scheduling the sequences of I/O operations that must be executed through this node (hence of the applications mapped on the compute nodes associated to this I/O node).

We define several objectives. Given a schedule, each job J_i is released at time r_i and finishes its execution at time C_i.

The *stretch* ρ_i of J_i is the ratio between the minimal execution time and the actual execution time:

$$\rho_i = \frac{\sum_{j \leq n_i} a_{i,j} + \frac{b_{i,j}}{b}}{C_i - r_i} \tag{III.2}$$

(where b is the maximum available I/O bandwidth). A stretch of 1 means that the application is not impacted by the other applications running on the system. A stretch of 2 means that due to I/O contention, the application takes twice as long to execute as it would normally. Typically the stretch is an objective more *user* oriented.

Recall that the *makespan* $C_{\max}$ of a schedule is given as the end of the last execution:

$$C_{\max} = \max_i C_i \tag{II.4}$$

Typically, the makespan is an objective more *platform* oriented: with a fixed amount of work, the work over the makespan is the platform utilization.

Finally, our general optimization problem is the following. Given a set of jobs $J_1, \ldots, J_n$ and a platform with R I/O nodes, each with a bandwidth b to the PFS, and connected to P compute nodes. Find a schedule that minimizes either the total makespan, or that minimizes the maximum stretch ($\max_i \rho_i$). We call the general setup of these problems HPC-IO, and, depending on the function to optimize: MS-HPC-IO or ρ-HPC-IO.

When the number of I/O nodes is equal to 1, this reduces to finding the right allocation of I/O for the different applications. We call this subproblem IO-SCHED. Given an allocation to a solution of HPC-IO, one can compute then independently the I/O scheduling solution using an algorithm to IO-SCHED.

Note that both the HPC-IO problem and the IO-SCHED problems are NP-hard: HPC-IO easily reduces to the multi-processor scheduling problem; IO-SCHED has been shown to be NP-hard by Gainaru et al. [29].

III.C Pack scheduling to solve HPC-IO

Here, we focus on a special type of solutions to HPC-IO, specifically *Pack scheduling* algorithms. In Pack scheduling [10], the jobs are partitioned into series of packs, which are then executed consecutively. Tasks within each pack are scheduled concurrently and a pack cannot start until all tasks in the previous pack have completed (see Figure III.3).

Pack scheduling has been advocated [28, 61] as it provides an easier and flexible mean of designing and implementing novel algorithms while providing significant savings.

In this section we discuss some strategies for IO-SCHED and HPC-IO.

III.C.1 Policies for IO-SCHED with a single pack.

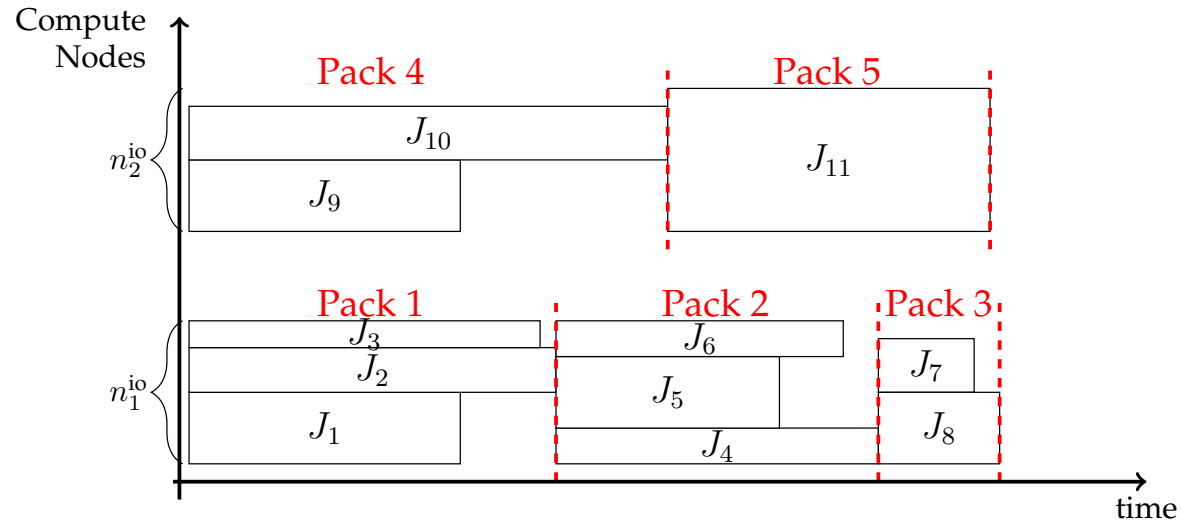
Once packs have been computed, we need to schedule applications within a pack. Several works have considered the problem IO-SCHED. In general there are two approaches list-scheduling heuristics [29, 84], or more involved pattern-based algorithms taking into account structural knowledge of the applications (such as their periodicity) [9].

In this work, we focus on list-scheduling based solutions. *List scheduling* policies consists in scheduling available I/O operations following a priority order as soon as a resource is available. They are an interesting way to solve IO-SCHED given the recent results in the previous chapter:

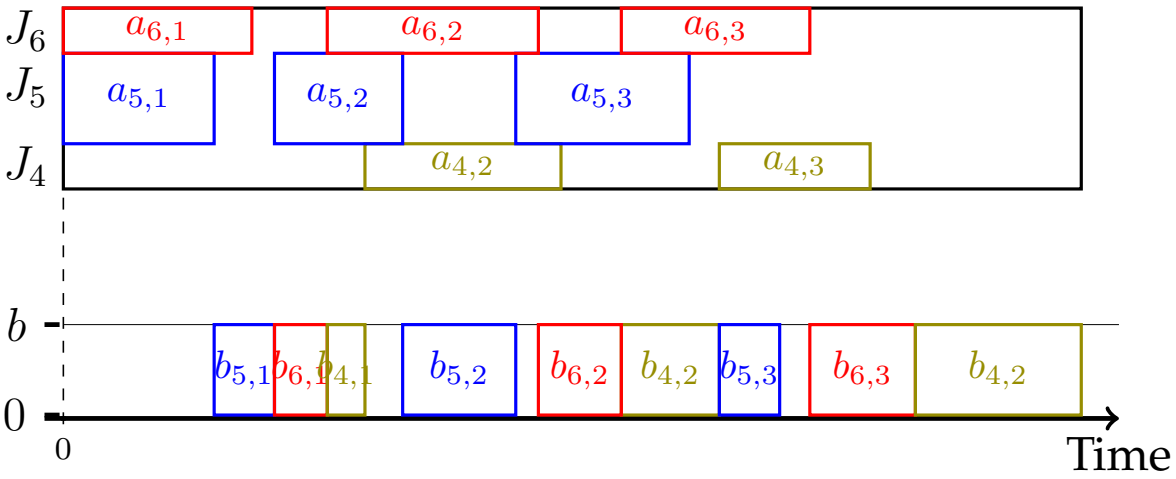
Theorem 3. *Any list-scheduling algorithm is a 2-approximation for MS-*HPC-IO *and this ratio is tight.*

Specifically, we consider the following natural priority orders that gives 8 different list-scheduling I/O policies:

1. Lowest ID: the scheduler picks the application with lowest system ID amongst those which requested I/O.
2. Longest I/O: the scheduler picks the application which performs the longest I/O phases (resource occupation).
3. Shortest I/O: the scheduler picks the application which performs the shortest I/O phases (avoid long wait).
4. Shortest remaining: the scheduler picks the application which is expected to have the least remaining work to do (free the machine ASAP).



(a) The mapping of 11 applications into packs for two I/O nodes



(b) The scheduling of I/O operations (bottom) within Pack 2. Computations are allocated on dedicated compute nodes and can start as soon as the I/O is transfered, but I/O operation share the available bandwidth b and can delay applications.

Figure III.3: A solution to HPC-IO of eleven applications on a machine with two I/O nodes.

5. Longest remaining: the scheduler picks the application which is expected to have the most remaining work to do (platform occupation).

6. FIFO: Applications are sorted by increasing I/O request time.

7. Bandwidth oriented: applications with the lowest ratio between I/O time and execution are advantaged (fairness).

8. Stretch oriented: applications with the worse stretch are scheduled first (fairness).

III.C.2 Algorithms for the mapping problem

In this section we provide an algorithm for the mapping problem that takes into account both the need for bandwidth and processor sharing between applications allocated to an I/O node.

The algorithm works in two steps: a first step partition the jobs into packs, while the second step schedules the pack on the different I/O nodes. The intuition of the partitioning algorithm is the following: we sort them by decreasing no contention execution time (as given by Equation III.1). Then we create packs following a *Best-Fit* procedure, that is a procedure that schedules greedily the next application in the first pack where it would fit with respect to an I/O constraint and a processor constraint, and otherwise that creates a new pack.

Precisely, to account this constraints, given a pack of jobs `Pack`, we define:

- $P^{\texttt{Pack}} = \sum_{i \in \texttt{Pack}} Q_i$, the number of processors used by the pack;
- $T^{\texttt{Pack}} = \max_{i \in \texttt{Pack}} \sum_{j \leq n_i} a_{i,j} + \frac{b_{i,j}}{b}$, the minimal length of the pack;
- $L^{\texttt{Pack}} = \frac{1}{b \cdot T^{\texttt{Pack}}} \sum_{i \in \texttt{Pack}} \sum_{j \leq n_i} b_{i,j}$, the average I/O occupation of the pack.
- S the *I/O sensibility* for pack creation. It is a parameter of the algorithm, by default $S = 1$.

The processor constraint to be respected is: $P^{\texttt{Pack}} \leq P$. The I/O constraint is $L^{\texttt{Pack}} \leq S$. Intuitively, this constraint tries to ensure that there is enough I/O bandwidth to perform all I/O operations with minimal delay.

We formalize this in Algorithm 2.

III.D Evaluation

In this Section we present experimental evaluations of both I/O scheduling and node mapping algorithms.

The evaluation is performed in several steps: in Section III.D.1 we study the impact of various I/O scheduling policies on IO-SCHED when the applications are already mapped on the machine. Then we study the impact of the mapping algorithm in Section III.D.2 on HPC-IO.

All experiments have been performed on the Tucan cluster consisting of compute nodes with Intel(R) Xeon(R) E7 with 12 cores and 128GB of RAM, interconnected with a 10 Gbps Ethernet.

Algorithm 2 Pack-partitioning algorithm.

1: **procedure** MAKE-PACK($J_i = (\prod_{j \leq n_i} A_{i,j}, B_{i,j})$) ▷ `Assume the jobs are sorted in decreasing order of` $T_i(B) = \sum_{j \leq n_i} a_{i,j} + \frac{b_{i,j}}{B}$;
2: Let S_P be a set of packs sorted by decreasing value of $P^{\texttt{Pack}}$ (empty initially);
3: **for** $i = 1$ to n **do**
4: Find the first `Pack` in S_P such that:

$$\sum_{j \leq n_i} b_{i,j} \leq (S - L^{\texttt{Pack}}) \cdot bT^{\texttt{Pack}} \tag{III.3}$$

$$Q_i \leq P - P^{\texttt{Pack}} \tag{III.4}$$

5: Fit J_i in this pack;
6: If there is no such pack, create an empty pack
7: **end for**
8: **return** S_P
9: **end procedure**

III.D.1 Impact of list-scheduling policies on IO-SCHED on a real machine

III.D.1.a Experimental setup

The workload is composed of five synthetic applications based on Jacobi decomposition and having different I/O, computational needs and number of iterations. Each application is configured to perform the CPU and I/O phases with a particular intensity level, that produces a different duration of the phases. The total execution load[1] is the same for all the applications. This means that if each of them would be executed exclusively their duration would be similar. All these applications have been executed in the proposed framework depicted in Figure III.1. This means that the application I/O phases are intercepted and coordinated by CLARISSE according to the I/O scheduling policy. The workload depicted in Table III.1.

III.D.1.b Result analysis and discussion

To see if we attain a steady state in terms of stretch and makespan we have run the workload described in table III.1 while increasing its number of *iterations batches*[2]. In Figure III.4, we plot the stretch of the different strategies when varying the number of iteration batches. We see that after some iterations the performance converges towards a steady state. In general, applications 3 and 4 are the ones who have the largest stretch.

[1] Number of iterations × (computational Intensity + I/O Duration)

[2] An iteration batch is a multiplicative factor that increases of reduces the application iteration number. For instance, a value of 2, makes the application run the double number of iterations (for instance, 500 for application 1).

Application id	Matrix size	CPU duration (s)	I/O duration (s)	Iterations
1	5000	32	10	250
2	5000	16	5	500
3	5000	8	2.5	1000
4	5000	8	2.5	1000
5	5000	16	5	500

Table III.1: Workload description for the experiment on the Tucan machine (for Figure III.4 to III.7).

This is explained by the fact that these applications have the lowest computational intensity and hence are more sensible to I/O delay. We can also notice that some policies have a large discrepancy in terms of stretch (for instance "longest I/O" or the "shortest remaining"). They exhibit applications with a stretch close to one (i.e. not delayed) and others with a very high stretch (up to 1.8). This is typical of a starvation situation where some applications are highly favored while others are able to perform I/O only when no other one is requesting access.

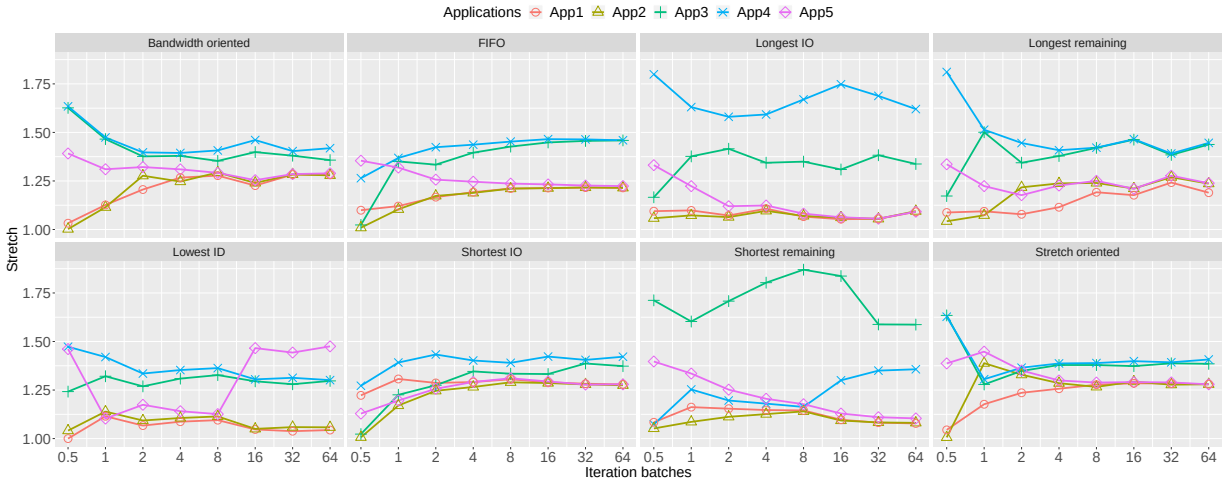


Figure III.4: Stretch of the different policies when varying the number of iteration batches.

In Figure III.5, we plot, for all policies, the maximum stretch of all applications. The maximum stretch measures the worst case for all the applications. We see that the "longest I/O" or "shortest remaining" policies behave the worst as they tend to exhibit starvation behavior. On the other side, several heuristics have a better behavior than the FIFO policy, which is the basic policy for this problem. In particular heuristics favoring applications that have a high stretch or the lowest ratio I/O vs. execution time behave much better than FIFO. Indeed, these two policies tend to greedily improve the

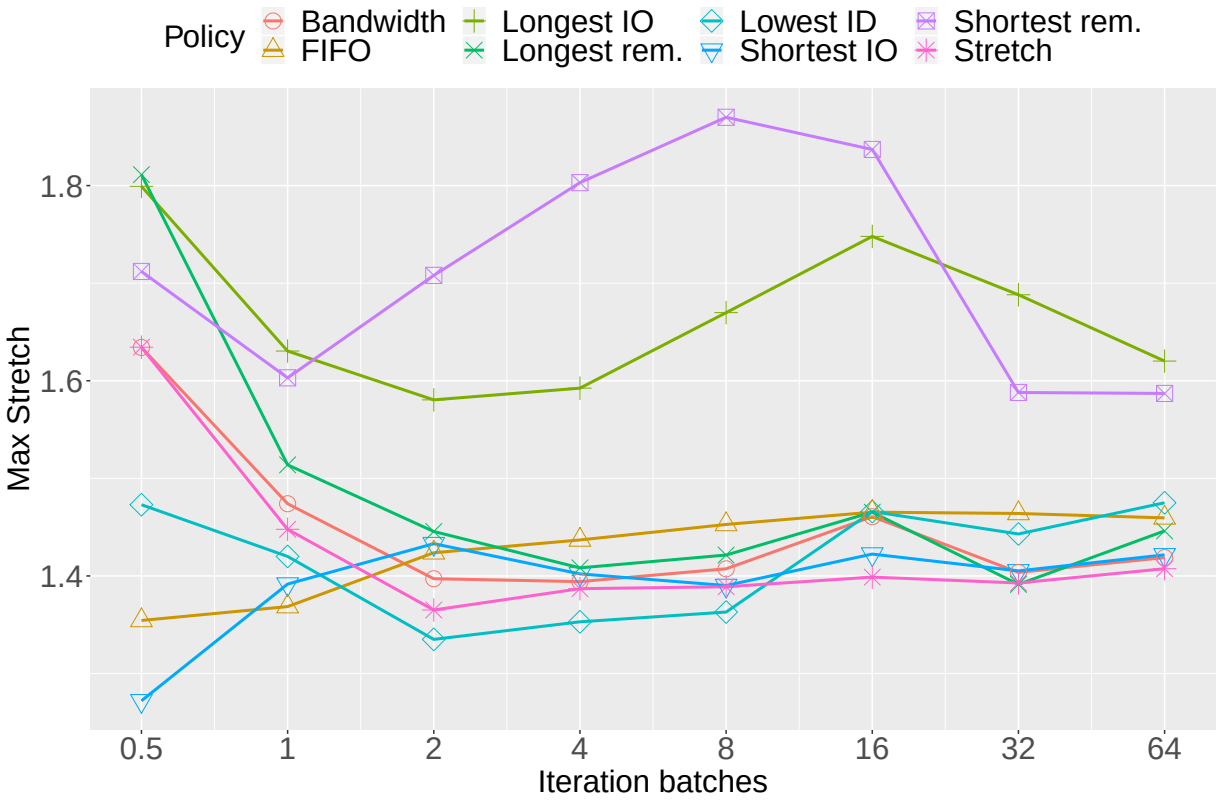


Figure III.5: Max Stretch of the different policies when varying the number of iteration batches.

.

fairness among the different applications.

In Figure III.6, we plot the makespan of all the policies compared to the default policy (FIFO). Surprisingly, we see that heuristics favoring the maximum stretch are also the ones that exhibit good performance in terms of makespan. Indeed, optimizing the maximum stretch requires to optimize resource utilization in order to avoid applications to be stalled. In addition, optimizing resource usage of each of them also optimize the time they spent in the system.

At the end, we see that the two policies that tend to sacrifice a resource in the short term behave poorly. Indeed, the "longest I/O" policy favors I/O and "Shortest remaining" favors computation: for all these cases such unbalanced policies lead to very bad fairness among applications. On the other hand, balanced policies that take into account the I/O and the fairness offer very good results compared to the FIFO policy. This is especially the case for the "Stretch oriented", the "Shortest I/O" and the "longest remaining" strategies.

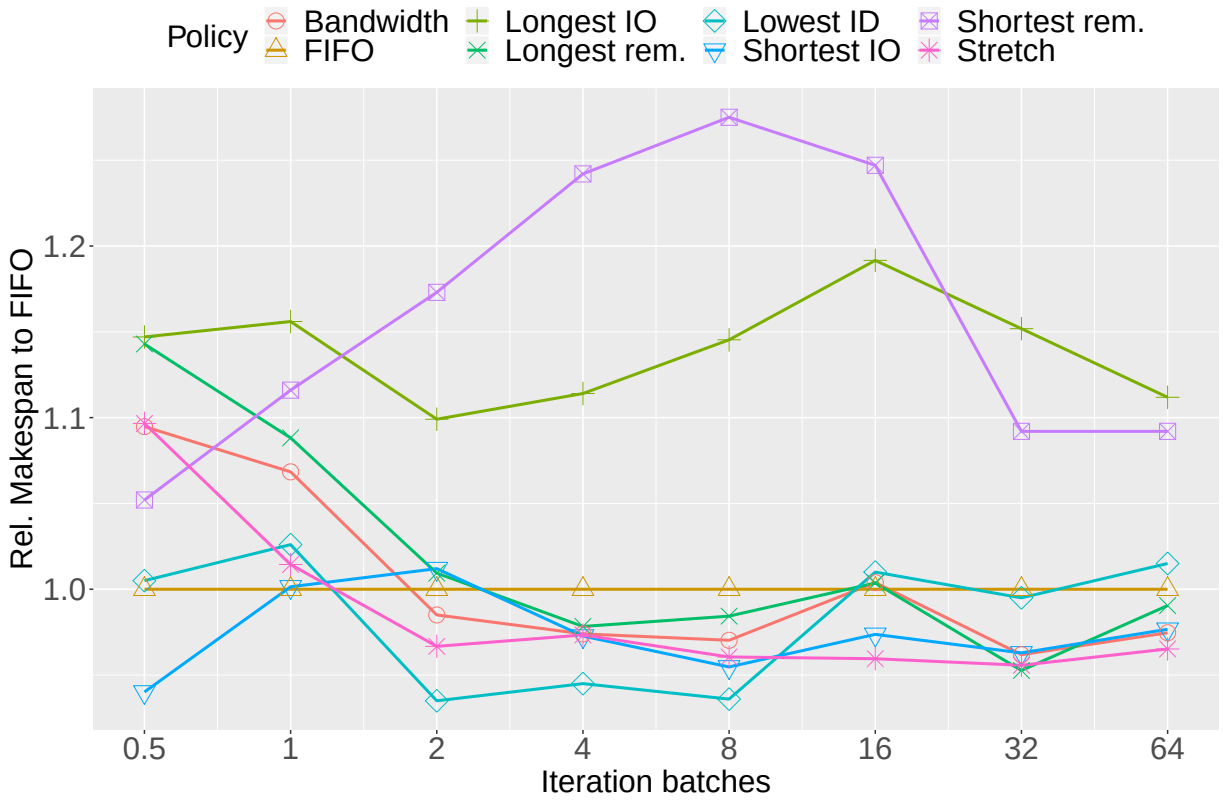


Figure III.6: Relative Makespan to FIFO of the different policies when varying the number of iteration batches.

III.D.2 Analysis of solutions for HPC-IO with emulation

In the previous section, we analyzed several I/O scheduling policies for the IO-SCHED problem. We now try to solve the more general HPC-IO problem by comparing the different mapping algorithms presented in Section III.C.2.

First we present the experimental setup (machine simulator and synthetic application generation). Then, the analysis takes place in two steps: first we study deeply the model with a single I/O node, then we extend it on a machine with multiple I/O nodes. Finally, in this section, we consider a machine with $P = 2048$ compute nodes per I/O nodes and one to five I/O nodes. We normalize the I/O bandwidth by setting $b = 1$.

III.D.2.a Machine emulation

To perform the analysis of this section at a larger scale (both in terms of number of applications and size of the target machine) we have designed a simulator able to test larger settings than those presented in Section III.D.1.

The simulator tool is integrated in the execution framework, which means that it is connected with the application scheduler as well as CLARISSE. Using the workload

provided by the scheduler, the simulator is able to compute the different CPU and I/O phases of each application by means of a fixed-increment time progression. In case of simultaneous I/O accesses, the CLARISSE's scheduling policies are used to determine the I/O access order. By means of this scheme, it is possible to simulate a given workload in the execution framework reusing the same software logic as the one used in the actual workload execution.

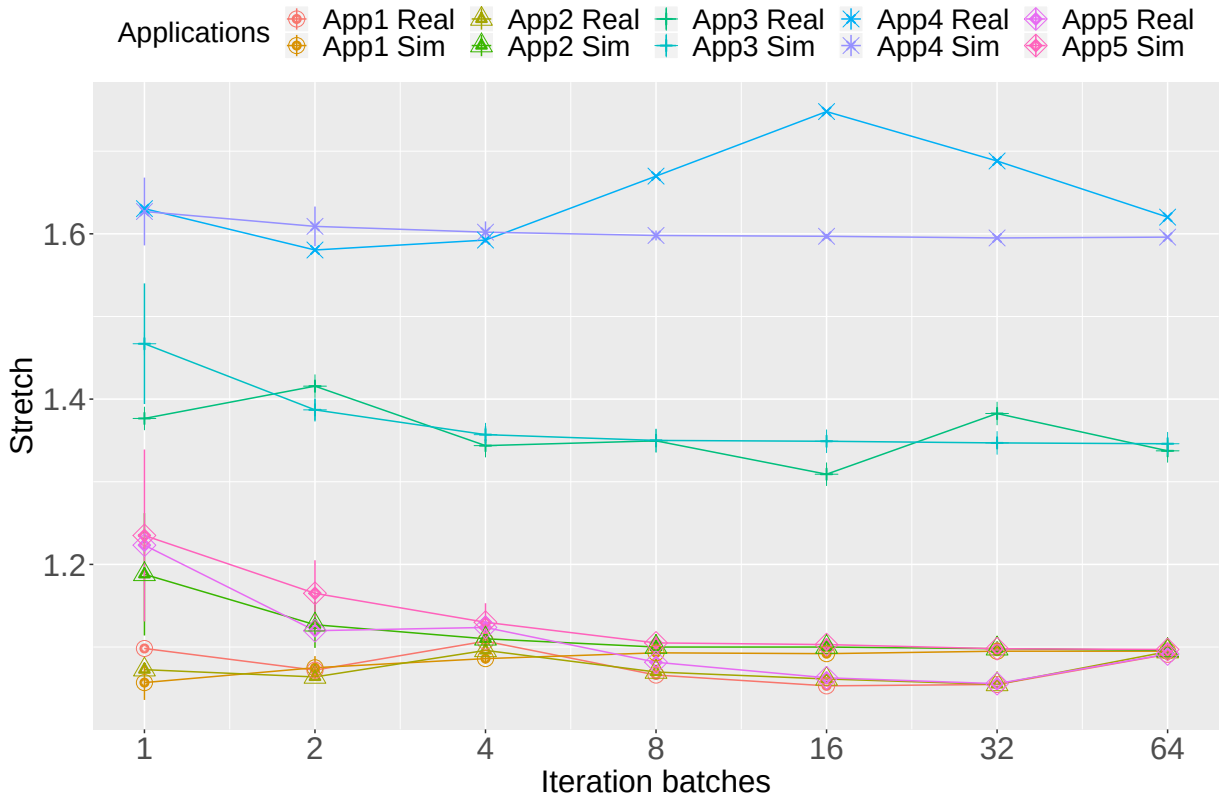


Figure III.7: Comparison of Simulated vs real execution (on the Tucan Machine) of the Longest I/O policy

Simulator validation In order to validate the simulator, we have rerun the experiments presented in Figure III.4. We have compared the results on both cases (simulator vs. real machine: Tucan). We have observed that for all heuristics the order between applications, for the stretch and the makespan, is kept. Moreover quantitatively the values are extremely similar: the largest difference between the simulation and the real execution, for 64 batches is 15%, with a geometric mean less than 5%. We give the case for the Longest I/O policy in Figure III.7 as it is a heuristic that displays high discrepancy in terms of Stretch.

III.D.2.b Synthetic workload generation

To perform the evaluation at a large scale, we propose a protocol to generate different and numerous workloads. We based the design on our protocol on two elements:

- The generated workload needed to be representative of I/O behaviors (hence including applications with high I/O load as well as applications with low I/O load).

- Then, we intuited that the impact of the different algorithm was correlated to a general property of the concurrent applications, namely the *average I/O occupation*. As seen in Section III.C.2 ($L^{\texttt{Pack}}$), this property is linked to the schedule as it takes into account the makespan. For the workload generation, we use a theoretical upper-bound for the average I/O occupation that assumes that there is no gap creating by the schedule. Mathematically, this writes as:

$$\alpha = \frac{P. \sum_{i=1}^{n} \sum_{j \leq n_i} b_{i,j}/B}{\sum_{i=1}^{n} (Q_i \cdot T_i(B))} \tag{III.5}$$

Based on these two preliminary elements, for each value $\alpha_{\text{gen}} \in \{0.5, 0.75, 1 \dots 10\}$, we generate 10 workloads in the following way:

- We pick the proportion β of applications with low I/O load at random on $[0, 1]$ (0 meaning all applications have high I/O load, 1 meaning they all have low I/O load).

- We generate the number of applications of each workload uniformly at random in $[25, 100]$.

Then for each application, for simplicity we assume that they are periodic (i.e. for all $j \leq n_i$, $b_{i,j} = b_i$ and $a_{i,j} = a_i$), and:

- Their number of iterations n_i is chosen uniformly at random between 250 and 1000.

- a_i is chosen uniformly at random in $[10, 100]$.

- Applications I/O load (b_i/a_i, and ultimately b_i) is chosen:

  1. Following a normal distribution of mean $\mu_1 = 0.1$, variance $\sigma_1 = 0.1$ truncated on the interval $[x, y]$ for applications of low I/O load;
  2. Following a normal distribution of mean $\mu_2 = 0.9$, variance $\sigma_2 = 0.1$ truncated on the interval $[x, y]$ for applications of high I/O load.

- Finally, to compute the number of processors Q_i of each application, we use a distribution in the discrete set $\{2^j\}_{j=0...11}$ of mean[3]

$$\bar{Q} = \frac{P(\beta\mu_1 + (1-\beta)\mu_2)}{\alpha_{\text{gen}}(1 + \beta\mu_1 + (1-\beta)\mu_2)}.$$

This protocol ensures that we have a various set of workloads, covering different I/O load (α). Note that in the rest of the evaluation, for each workload, we use their actual I/O load as defined by Equation (III.5), and not the value of α_{gen} used for the generation. The precise code for the generation and execution of the workload is available at https://gitlab.arcos.inf.uc3m.es:8380/desingh/IOscheduling.git.

III.D.2.c Evaluation with a single I/O node

In this section we study the overall problem HPC-IO and its solutions consisting in (i) the Make-Pack procedure (Sec. III.C.2) including its sensitivity parameter S; (ii) once the packs are done, the different I/O policies.

The experiments are done with comparison to a baseline algorithm: First-Fit [10] for pack creation and FIFO for I/O policy (shown to be the most effective policy in Section III.D.1). These two strategies are globally referred to as *First-Fit* in the following. As the baseline algorithm, the pack creation does not take into account the I/O operations of jobs but only their execution time when performed by themselves on the machine. One can observe that essentially this is the Make-Pack procedure when the sensibility $S = \infty$.

In this Section, we study two different pack creation algorithms:

- Make-Pack when the sensibility is set to $S = 1$ (referred to as *Sensibility=1* in the following): intuitively, this strategy tries to minimize the likelihood that there are delays due to I/O.

- Make-Pack when the sensibility is set to $S = \alpha$ (referred to as *Sensibility=I/O load* in the following). This is intended as a middle-case behavior that one would obtain if S varied.

Finally, we emulate the execution of the packs using the simulator described in Section III.D.2.a using a defined scheduling policy. As default scheduling policy, we use FIFO for both pack creation algorithms. In addition, we have also performed experiments with *I/O Sensibility=1* and with the "Longest remaining" policy which was proven to be very effective in Section III.D.1.

[3] $\bar{Q}$ is obtained by replacing in Equation (III.5) all values by their average value, which is not mathematically correct but that we use as a first approximation to generate the workload.

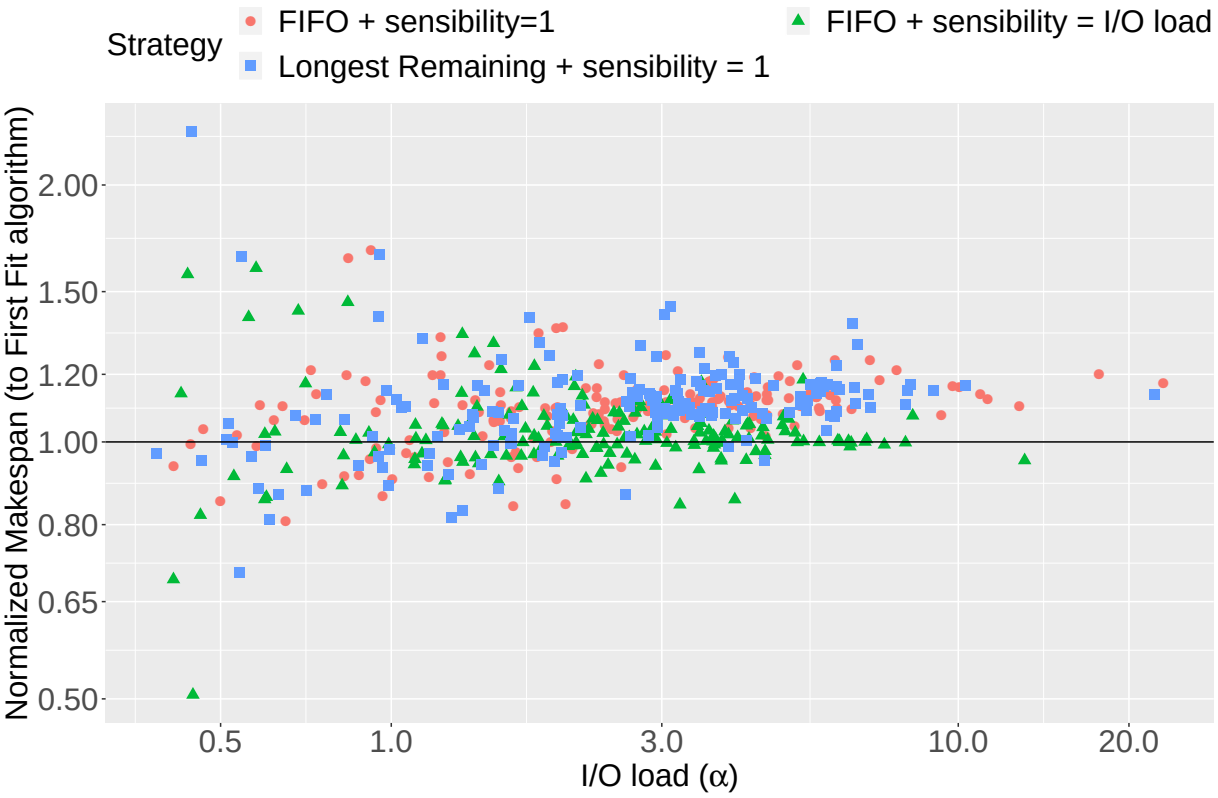


Figure III.8: Comparison of makespan for different strategies

Overall performance To study the performance, we study both makespan (Figure III.8) and the stretch[4] (Figure III.9) of the different solutions. The results are presented normalized to the respective performance of First-Fit, and we study them as a function of the I/O load α (Equation (III.5)).

In the standard configuration, with a FIFO scheduling policy and a Pack partitioning sensibility equal to 1, Figure III.8 shows an average 10% overhead of our algorithm in terms of makespan while Fig: III.9 show significant stretch improvement. The significant stretch improvement could be expected: with the sensibility set to 1 we reduce contention a lot, and intuitively our stretch stays close to 1. On the contrary, First-Fit has an average I/O occupation factor that increases potentially with α, hence increasing the contention and the stretch.

Changing I/O scheduling policy does not seem to have an impact on these measures. Hence in the next evaluations we only consider FIFO policy.

It was to be expected as bandwidth-aware heuristics produce more packs. Indeed, since we add more constraints on the packs (an I/O constraint, Eq (III.3)), when this constraint is saturated and if it occurs before the processor constraint (Eq (III.4)), new packs are created. Figure III.10 shows how many more packs are produced by our al-

[4]In this section, we use the average of the maximum stretch of each pack.

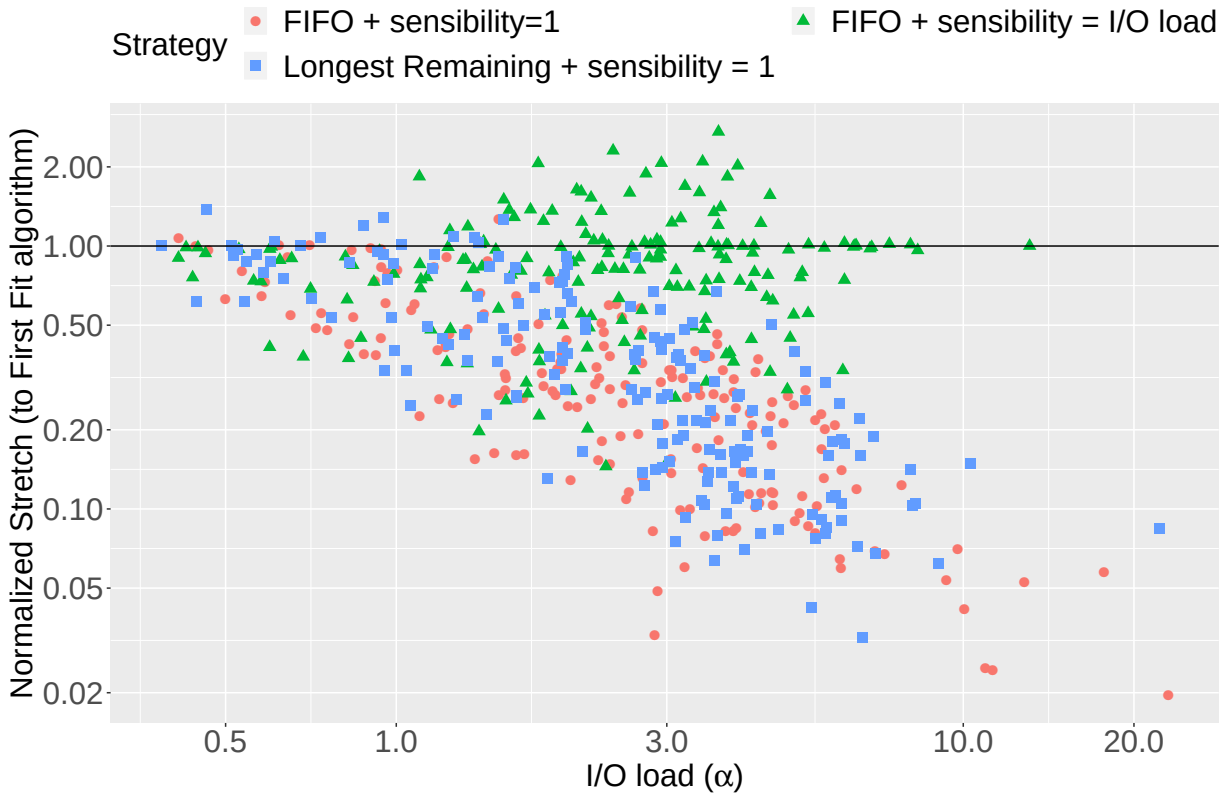


Figure III.9: Comparison of stretch for different strategies

gorithm compared to the First-Fit case when varying the I/O load. We see that, for a sensibility of one, the number of packs is increasing with the I/O load. This is due to the fact that when the I/O load is large the Pack Partitioning algorithm creates more pack to avoid I/O contention. When the sensibility is α, the ratio is much smaller and roughly constant because the sensibility determines how contention is avoided. Producing more packs has the advantage, however, of decreasing contention, which explains the improvement in stretch. The downside is increasing the number of packs and unused processors within packs.

As the stretch improvement is noticeable, we may consider that a sensibility of 1 for Pack Partitioning Algorithm is too pessimistic and leads to an unbalanced trade-off. Increasing the bandwidth limit as a function of α provides an alternative compromised, mitigating the makespan loss while maintaining stretch improvement most of the time.

In depth performance For an in-depth performance evaluation, we focus on the FIFO policies.

We present in Figure III.11 the ratio of the execution time as measured to an ideal one that one could predict with no contention would occur (essentially if the global I/O bandwidth was unbounded but keeping the individual I/O bandwidth bounded).

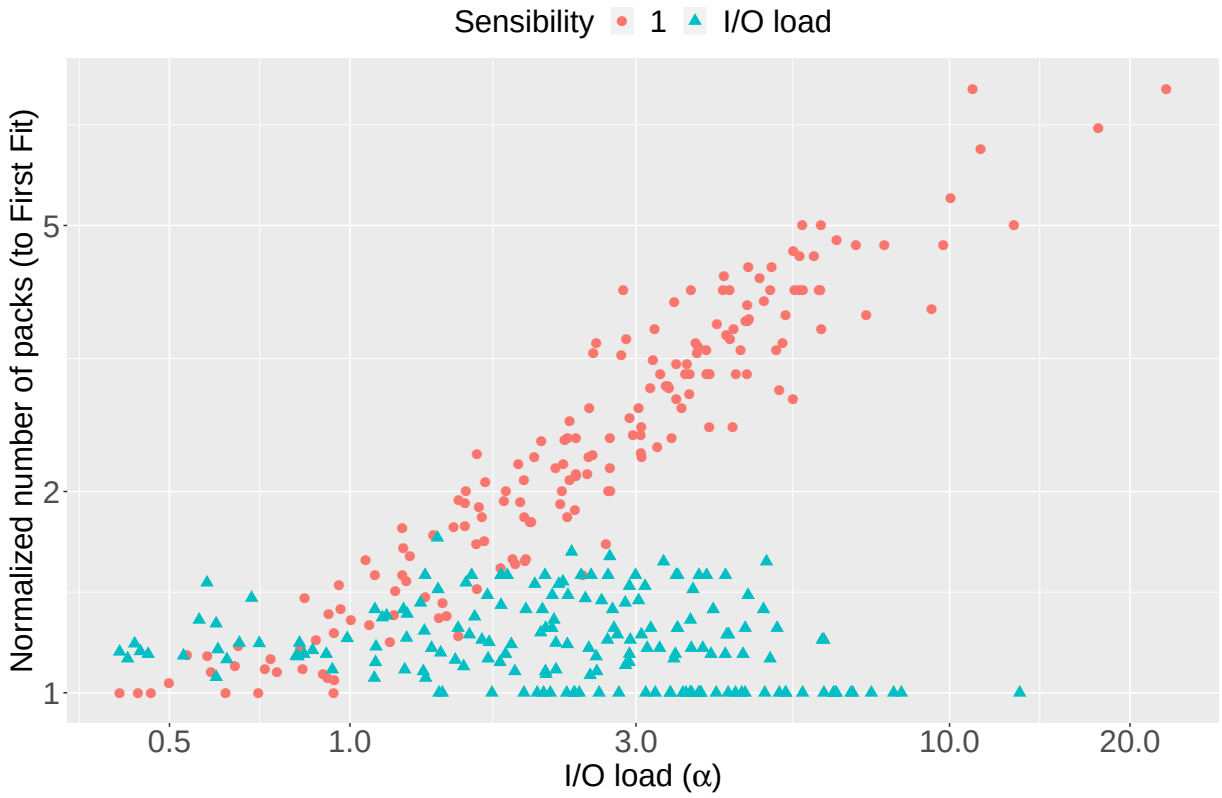


Figure III.10: Normalized number of packs produced by the Pack Partitioning algorithm (relative to the First-Fit algorithm) for different sensibility

The interesting observation is that the execution of Pack Part ($S = 1$) is a lot closer to the ideal one (within 20%, while First-Fit can be as bad as 1000%). This provides more control on the execution. This is coherent with the results observed by Herbein et al. [36]. The version of Pack Part with $S = \alpha$ is close in behavior to that of First-Fit as one would expect since the congestion constraint is relaxed.

This results is even more interesting when we study for each compute node the average time that they spend: (i) doing *useful* execution (either they are doing some compute, or the application mapped on them is performing I/O), (ii) being delayed (the application mapped on them is waiting to perform I/O), or (iii) being idle (there is no application mapped on them, or the application mapped on them has finished working and the pack is waiting for some final applications). We plot these average time in Figure III.12, normalized with respect to the average time of First-Fit, so that, if the input to all algorithms were identical the *Exec* time (useful execution time) would be identical. Here, the difference in *Exec* time for the three algorithms is an attribute of the randomness in the generation of workloads.

This figure is interesting because again, we observe that for a trade-off of 10% in makespan there is a transfer of 25% of the time from delay to idle. In addition, the ad-

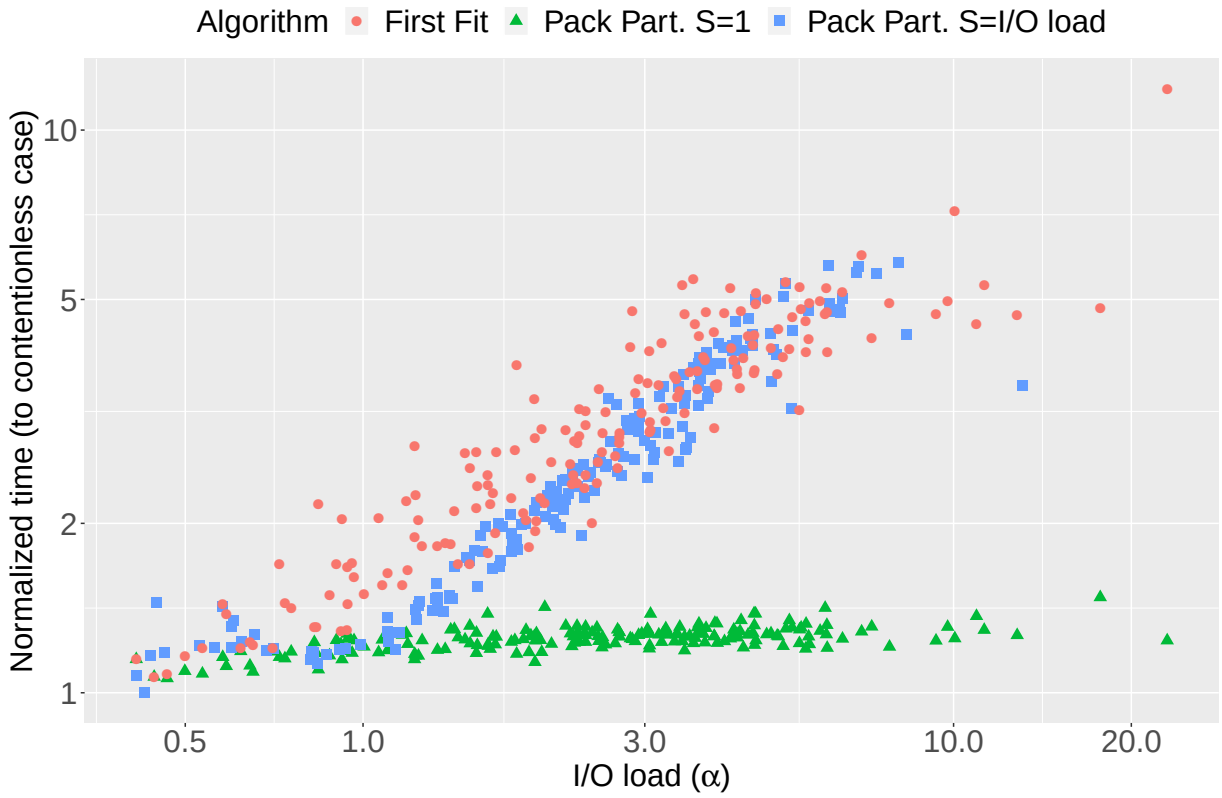


Figure III.11: Processor time and robustness: pack algorithm vs. First-Fit

ditional 10% of time wasted is also moved to idle time. This gives other opportunities (such as turning-off nodes for energy consumption, using available nodes for backfilling operations with applications that do not need I/O etc), while the idle time due to contention is lost. This is another strong argument for bandwidth-aware scheduling policies, even if locally it reduces machine utilization, globally it provides an opportunity to improve it by a lot more than what is wasted.

III.D.2.d Multiple I/O nodes

Previous experiments were done using only one partition the compute nodes with one I/O node: packs were executed sequentially. However, in many platforms, several I/O nodes belonging to different partitions/racks are available (see Figure III.2). In order to see the impact of this feature, we then studied the parallel execution of the computed packs.

To do so, given the assumption that I/O nodes do not interact with each other and that they have the same characteristics, we can simply allocate the packs computed by Algorithm 2. In this section we only consider the case where the sensibility $S = 1$, and we use FIFO as the default I/O policy. The pack allocation is done using the Largest

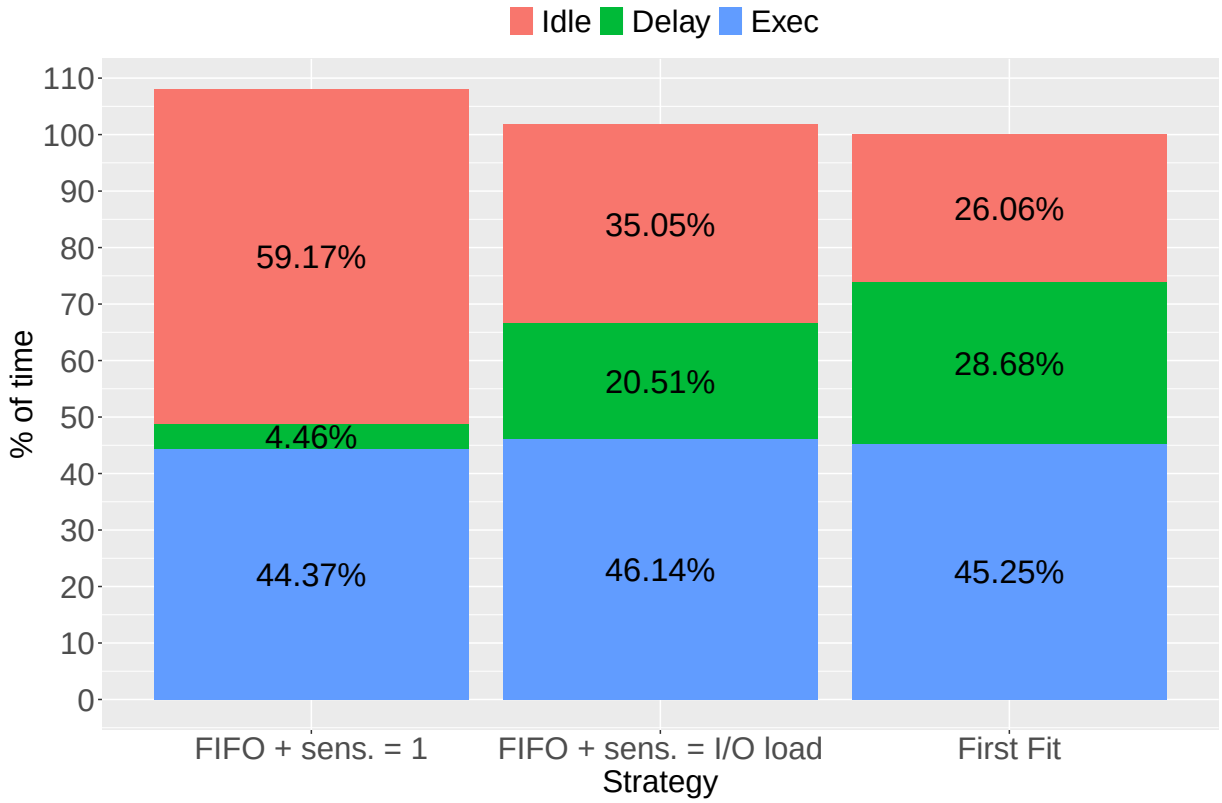


Figure III.12: Average execution, idle and delay time (normalized) for different strategies

Processing Time[5] (LPT) heuristic, where pack duration is the contention less execution time of its longest application.

Here, we use the FIFO I/O scheduling policy and the standard configuration of the Pack Partitioning algorithm (sensibility=1). Hence, the one I/O node case is the same as the *sensibility = 1* case of Figure III.8.

We see that, when we increase the number of I/O nodes the relative makespan is decreasing. The geometric mean[6] of the 1 (resp. 3 and 5) I/O node(s) case is 1.09 (resp. 0.71 and 0.53). This means that in the five I/O nodes case our bandwidth-aware solution is, on average, twice as fast as the First-Fit algorithm!

The interpretation of this result is the following. With our Pack Partitioning, we have more but smaller packs than for the First-Fit case (see Figure III.10). Hence, providing a balanced allocation is easier in this case than for the First-Fit case where packs are less numerous but longer. Moreover, we are computing the pack allocation based on the estimated pack duration ignoring the contention. These durations, as shown in Figure III.11, are more precise in the Pack Partitioning case than in the First-Fit case. Hence,

[5]We map the longest remaining pack on the I/O node on which it will finish the earliest
[6]We use the geometric mean instead of the arithmetic mean as we are dealing with ratios

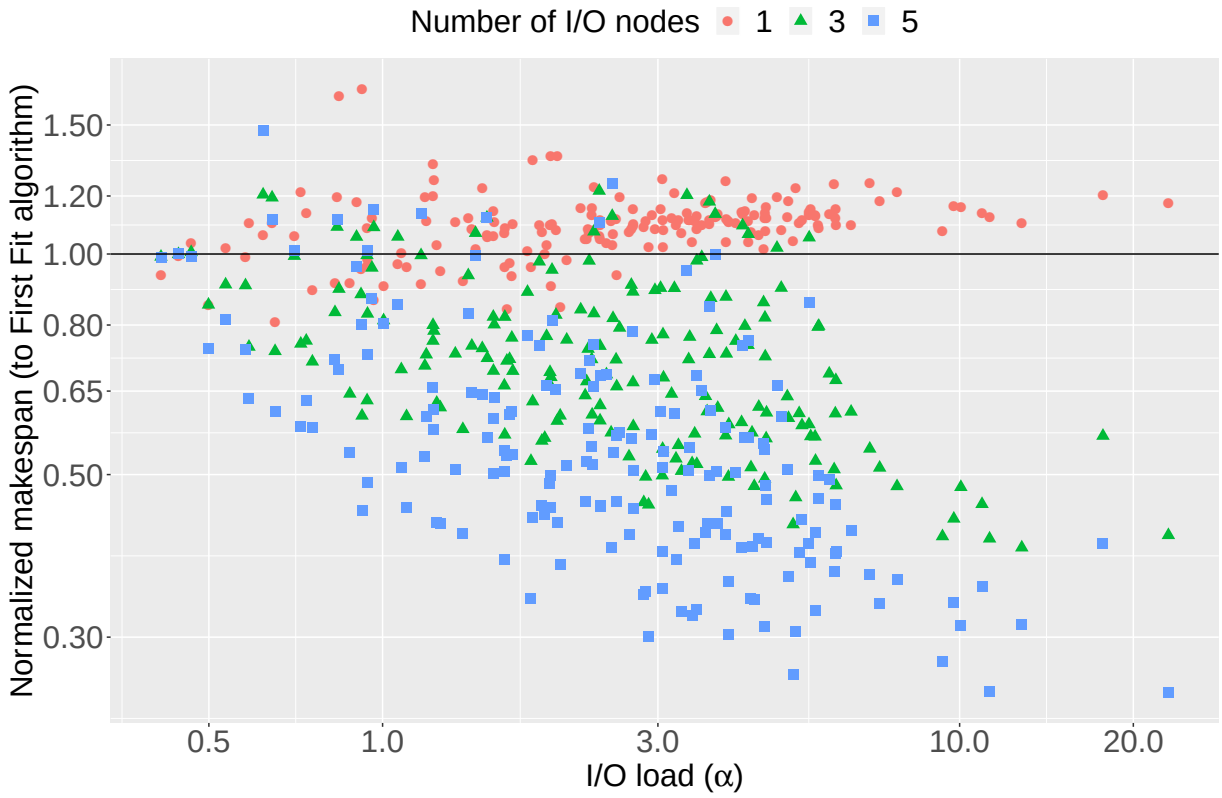


Figure III.13: Comparison of the relative makespan of Pack Partitioning Algorithm (with sensibility =1) to the First-Fit algorithm with multiples I/O nodes.

the load balancing computed is more accurate with our Pack Partitioning solution: allocation decisions are more robust and hence lead to better solutions.

Last, the I/O nodes are homogeneous and the stretch is a local metric of each individual pack. Therefore, the stretch performance of all algorithms does not depend on the number of I/O nodes and is exactly the same as the one depicted in Figure III.9 for a single I/O node.

In conclusion, in a realistic setting where there are multiple I/O nodes, our Pack Partitioning algorithm outperforms the First-Fit algorithm both for the stretch and the makespan.

III.E Conclusion and prospects

This chapter addressed the issue of executing concurrent applications in a system with bounded I/O bandwidth. We presented a model, optimization framework as well as simple heuristics for the issue of allocating the applications depending on the I/O resources available on the machine. Through rigorous experiments, our first observation was that simple and fair list-scheduling policies seemed to perform better in the long

run when it comes to scheduling I/O access. In addition, we presented a simple strategy to allocate applications together based on an approximation of their resource usage. Our evaluation of this strategy gave interesting results: with a single I/O node, they improved importantly the stretch of the machine while degrading slightly the makespan (or throughput). But a more in-depth study showed that this degradation was a consequence of a much better control of the waste (mostly by having more unoccupied resources instead of resources waiting).

We have also studied the case where compute nodes are decomposed in several partitions/racks by increasing the number of I/O nodes. In this case our bandwidth-aware strategy performs better for both metrics.

After this chapter, several directions open up, both for the mapping of applications and the scheduling of I/O. With respect to the mapping, our natural first step will be to remove the pack constraint and see if we can design an efficient bandwidth-aware strategy. Another direction would be to simply enriching this Pack Partitioning policy with small job backfilling. Both these directions are discussed in Chapter IV. We spent some time trying to define how measure rendering of a schedule quality in regard for backfilling. Such a measure would meter the amount of unused resources and for how long. At equal performance, it is better to "regroup" the unused resources, having as many idle processor and as long as possible to enable easier re-allocation. Then, we should move from exclusive-access I/O policies to policies where the bandwidth can be shared is a direction we intend to take, we will initiate a discussion on this matter in Chapter V. The difficulty here is to make the middleware ready to evaluate these strategies, maybe through containers.

This chapter marks a significant step in our thesis. First, it shows the difference in magnitude of the improvement we can gain while mapping the applications in comparison with only scheduling the accesses. Of course, this study is done based on a theoretical model and validated through simulation, without "real" I/O accesses so material limitations do not occur. This difference of magnitude is not unexpected either. Compared to scheduling that only change application behavior but does not change the platform occupation paradigm, mapping use it as a leverage for resource control. Necessary, this better resource control enables better parallelism.

Chapter IV

Comparison with List-Scheduling - Strict constraint vs on average

In this chapter, we discuss the impact of data-awareness on the design of resource management algorithms. Specifically, we focus on the incorporation of the I/O needs of applications, one of the current major bottlenecks in HPC systems, into the batch scheduling algorithms. Hence the targeted solutions should be (i) scalable (thus have a low complexity); and (ii) easy to implement and to test to be adopted. For this reason, we focus this study on two simple paradigms: list-scheduling and pack-scheduling (see Figure IV.2). We presented *Pack-Scheduling* and studied in the previous chapter. Pack-Scheduling is a strategy that maps applications by packs, i.e. sets of applications that start at the same time. The next pack cannot start as long as the last application of the previous pack has not finished its execution [68, 83]. *List-Scheduling* does not impose this constraint: it sorts the tasks given a priority order (typically, First-Come-First-Served in batch schedulers [3, 41]), and schedules them as soon as there are enough compute nodes available.

In this chapter, we discuss and compare IO-aware versions of these two paradigms. As in the previous chapter, for scalability we consider a two-pronged approach:

- The batch-scheduler uses average I/O information (which can be collected using tools such as Darshan [66]) to create the mapping;
- An I/O scheduling middleware (such as Clarisse [39]) then schedules the concurrent I/O using online heuristics.

In the same way as in chapters II and III, there are two objectives to account for: a platform-oriented objective (maximizing the utilization of the machine, i.e, the number of FLOP/s), and a user-oriented objective (fairness).

In this chapter, we present the following important contributions:

- While list-scheduling is generally the standard mapping heuristic for batch-schedulers, we show that in the case of I/O intensive workloads, an I/O-aware pack scheduling strategy may be more efficient;

- For Pack-scheduling, we show that a *characteristic time*, i.e. the order of magnitude of I/O transfers, can be taken into account to provide fairer policies. We provide intuition and discuss the implications of this.

- We underline the strong relation between the workloads and the scheduling policy relevance and provide insight on how to choose an adequate one.

IV.A Simulation

IV.A.1 Constraint implementation

As described in chapter III, we chose to model the I/O behavior of an application by summing all volumes of I/O transfer and dividing by the application execution duration. In the heuristics, bandwidth constraints are enforced based on this value in such a manner that the sum of all concurrent applications does not exceed the limitation. This way, we obtain a model where all applications have a fluid I/O behavior in opposition to their real-life, periodic one. These constraints can be implemented either as a strict limit as described in IV.A.2 or as an average one in IV.A.3. These different heuristics are then evaluated in realistic scenarios, with periodical I/O phases performed exclusively.

IV.A.2 List-scheduling (LS)

The first way to implement I/O limitation is to determine a *strict constraint* at every moment. The sum of the average I/O bandwidth of all applications running at a time cannot go over this threshold. Applications are scheduled in their order of appearance in the workload on a first-come, first-served basis as soon as both their processor and their bandwidth requirements are met.

IV.A.3 Pack-Scheduling

Chapter III, which resumes our article Carretero et al. [83], demonstrates that making packs of applications starting at the same time and running concurrently can improve the control over congestion. Here, we resume the same procedure. In Pack-scheduling, the I/O constraint is enforced *on average*. Indeed, the average I/O bandwidth of a pack is the sum of all I/O operations performed by the packs application divided by the pack duration. It means that I/O operations can go past the I/O threshold at some points of the execution. The pack heuristic was presented in Figure 2.

Variants Depending on the way we sort applications, we can change our objectives. In the following, we propose three different ways to build packs.

1. Without sorting applications (Random), this serves as a baseline. It can also describe a way to build packs in steady-state without an overview of the application pool.

2. Sorting applications following non-increasing execution time (Max), this is the default way to make optimized packs, as scheduling applications with similar duration minimizes the resource imbalance at the pack execution end.

3. Characteristic time (Char) is the duration of a period consisting of one compute phase followed by an I/O phase. Sorting applications by this parameter is a less intuitive approach. As I/O are performed exclusively, there is a delay induced every time a request is blocked by another operation. Based on the periodic behavior of applications, we expect the I/O phases to be delayed in the first iterations. Then, if the period span are close enough, a *synchronization* effect may occur and I/O can be performed in turn with little to no delays. A simple example is shown in Figure IV.1.

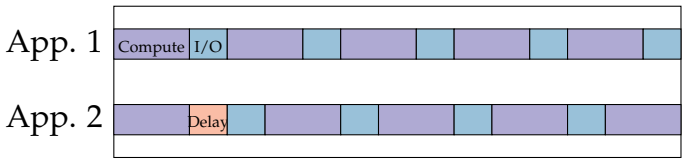


Figure IV.1: Synchronization example with two identical applications. I/O are performed in turn and no delay occurs past the first iteration.

IV.A.4 Backfilling

In pack scheduling, the platform nodes become idle during the execution of each pack as applications terminate. These nodes are not re-used until the next pack starts. In a minor extent, idle resources waiting for a future application can occur as well as in LS. However this performance loss can be avoided. Indeed, the packs and applications duration can be predicted as well as the amount of nodes involved. Using this information, we can see if an application fits in the free space and modify the schedule accordingly. In the following, such a backfilling strategy is implemented for both pack and list scheduling.

IV.B Evaluation Methodology

Tampering directly with a production system, impacting the work of users solely for the sake of evaluating prototype strategies is not an option. Evaluating the relevance of the aforementioned strategies and their consequences is thus particularly challenging

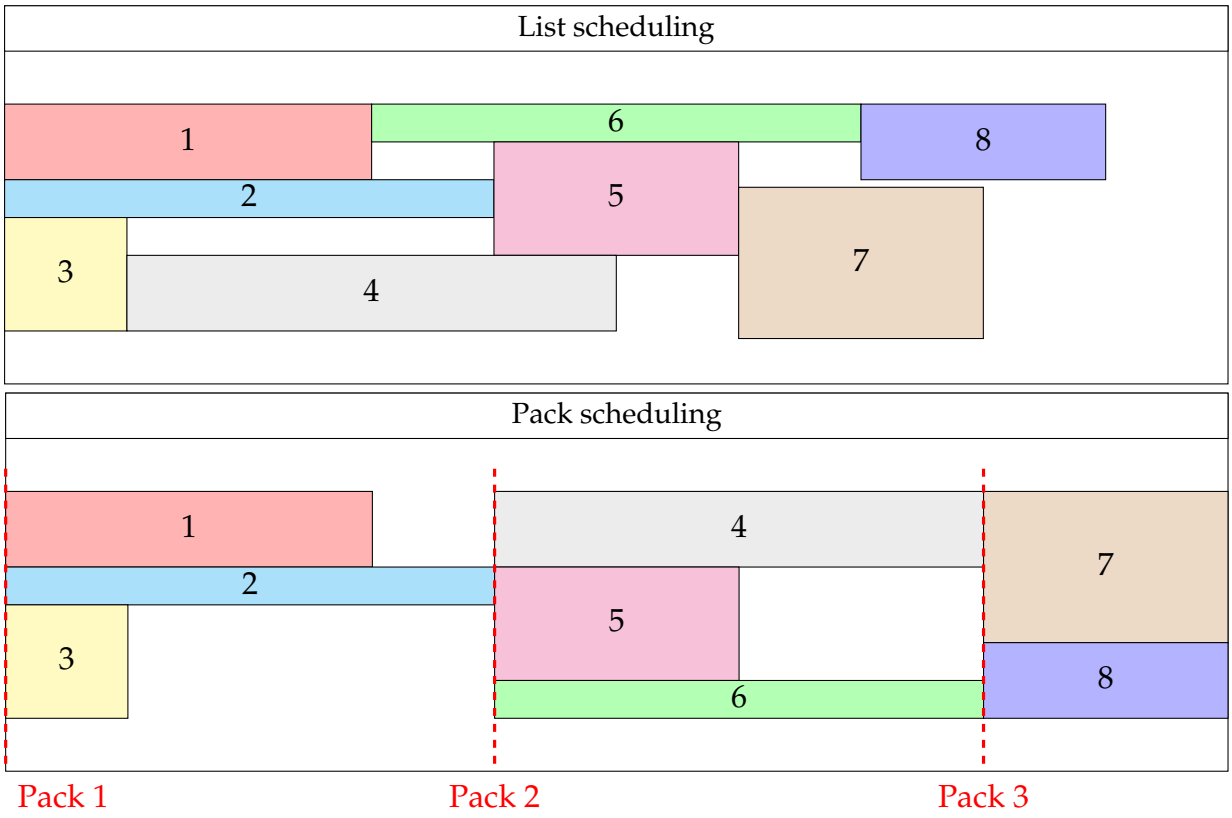


Figure IV.2: An example of list scheduling and pack scheduling for the same set of tasks (packs are separated by dotted red lines).

and hence requires, as a first step, the use of heavy simulation to compute statistically significant evaluations.

The coherence of the behavior of the simulator that we describe below and that of real machines were verified (with Vesta, a development platform for Mira in [29], and with Jupiter a Cluster at Mellanox in [30]) with IOR benchmarks. Hence, for this evaluation we will rely solely on this simulator which we detail below.

IV.B.1 Machine model for the simulator

The model used is similar to the one presented in the previous chapter and illustrated in Figure III.2. We recall its main characteristics hereafter.

In high performance computing, parallel platforms consist of computational resources structured in racks composed of compute nodes. One (or sometimes several) I/O node is available on each rack for the compute nodes to access the parallel file system (PFS). Each I/O node has a fixed, limited *I/O bandwidth* and hence compute nodes share this bandwidth when accessing data on the PFS. In many supercomputers racks are homogeneous: on each rack, the I/O bandwidth is the same as well as the number

of compute nodes associated to each I/O node.

Therefore, without, loss of generality, in this paper, we will consider only one rack with one I/O node: extending it to several racks (or I/O nodes) [83] is straightforward: the proposed solutions will be applicable to an arbitrary number of racks considering the platform homogeneity.

Moreover, for the evaluations, following the trend in current I/O management software (such as Clarisse [39]), we consider that simultaneous bandwidth sharing is not allowed (i.e. on a given I/O node, only one application is performing I/O at the same time). Blocking I/O guarantees that at all time the I/O bandwidth is not overloaded, hence we do not need here to model I/O congestion. We discuss I/O blocking and its consequences in the future work section. Additionally, we assume that I/O preemption is not allowed either: once an application has started to perform I/O, it has to finish its transfer.

IV.B.2 Applications

As in our previous chapters, we recall that our applications model considers a series of consecutive *non-overlapping* phases:

- compute phases (executed on the compute nodes);
- I/O phases (a transfer of a certain volume of I/O using the available I/O bandwidth) which can be either reads or writes.

In order to run properly, the applications must have access to sufficient computational and I/O resources. The amount of processors requested is defined beforehand by the user and can be seen as one of the application characteristics. The amount of data to be accessed during the I/O phase is also a characteristic that is assumed to be deterministic. We assume that the user is able to give a cumulative behavior.

Formally, we have a set of n jobs $\{J_1, \ldots, J_n\}$. Each job J_i requests Q_i compute nodes for its execution. J_i consists of n_i successive, blocking and non-overlapping operations:

- $A_{i,j}$ (a compute operation that lasts for a time $a_{i,j}$);
- $B_{i,j}$ (an I/O operation that consists in transferring a volume $b_{i,j}$ of data).

Therefore, if the bandwidth available to J_i to transfer its I/O to the PFS is equal to b, the time T_i needed for the total execution of J_i is:

$$T_i(b) = \sum_{j \leq n_i} a_{i,j} + \frac{b_{i,j}}{b}. \tag{III.1}$$

Synthetic workload generation

For the evaluation, we do not consider release time for the application but consider them to be released as a single batch. This will have implications in how we evaluate the performance of the algorithms which we discuss in the next section.

In order to perform a significant number of experiments, we designed a generator aiming to produce diverse fitting workloads.

Characterization We hypothesize that the impact of I/O restriction policies depends on the amount of I/O operations performed by the workload.

Therefore, in our study, we tested the impact of I/O scheduling policies as a function of the I/O load of the system. Given a workload, we recall that I/O load (or I/O intensity) definition, α, as an upper-bound on the I/O bandwidth exertion is given in equation (III.5):

$$\alpha = \frac{P. \sum_{i=1}^{n} \sum_{j \leq n_i} b_{i,j}/b}{\sum_{i=1}^{n} (Q_i \cdot T_i(b))} \tag{III.5}$$

Application sets are generated following two different protocols detailed below: (i) The uniform protocol in which all parameters are independent; and (ii) the Mira-based protocol, inspired by data [59] collected on the eponymous supercomputer.

Real-life applications have pseudo-periodic behavior [16, 24]. Since the point of interest is to study I/O conflicts on large workloads, the fluctuation in operation times does not seem particularly relevant. Hence, to simplify the model at hand we assume that each application is periodic (i.e. for all $j \leq n_i$, $b_{i,j} = b_i$ and $a_{i,j} = a_i$) in all generated workloads.

IV.B.3 Mira-based generation protocol

IV.B.3.a Processor repartition

Data found from Mira [59] shows that the required processors per job follow a discrete exponential distribution of parameter $\lambda = 1.35 \cdot 10^{-4}$ with values ranged from 512 to 49152.

IV.B.3.b Job duration

The same source shows that the job duration depends on the amount of processors used.

- Less than 4K nodes, a median job time of 1 hour
- Between 4K and 16K nodes, more than 2 hours
- More than 16K, 0.5 hour

In each case, we use normal distribution with the same median and a variance of 10 percent. These distributions are obviously reduced to their positive values.

IV.B.3.c Periodicity

Whereas papers emphasize the periodic aspect [16, 24] of jobs, they rely on the analysis of specific jobs.

The diverse IO phase occurrences are often described by their average amount per execution or their amount of function calls. Neither of these descriptions is suitable to describe applications following our theoretical models.

We have chosen to use a uniform distribution ranging from 10 to 100 iterations. The number of iterations only impacts the length of a period as the application duration is defined beforehand. Therefore if there are enough applications running, these parameters have little impact on the resource usage. However, for a fixed execution the more iterations there are, the closer we are to the continuous model and hence should behave as expected by the algorithm.

IV.B.3.d IO generation

We explained above that applications I/O are often measured using their amount of operations. It is not straightforward to deduce the amount of time spent performing I/O by a given application. Measurement produced by Gainaru et al. [29] on Intrepid seems to show that it follows then a uniform distribution with time ranging from 5 to 50 percent of the total execution time. However, this measurement takes into account the time spent while doing I/O requests and includes the time lost in contention.

We chose to draw the proportion of time spent in I/O for each application following (i) two normal distributions truncated to have values between 0 and 1, one having mean 0.1 for compute intensive applications, the other 0.9 for the I/O intensive ones. The prevalence of each depends on the workload expected I/O load. (ii) Alternatively, we use only one normal distribution centered on the expected I/O load.

In the following, we name these families of workloads, (i) *Mira binormal* (MB) and (ii) *Mira normal* (MN) respectively.

From this time, we compute the total I/O volume by weighting with the requested number of processors.

IV.B.4 Uniform generation protocol

- We choose an I/O intensity upper bound α_{gen}, as described in definition III.5 for workloads.
- We pick the proportion β of applications with low I/O load uniformly at random on $[0, 1]$ (0 meaning all applications have high I/O load, 1 meaning they all have low I/O load).
- We generate uniformly at random between 100 and 500 applications for each workload.

These applications have the following characteristics:

- Their number of iterations n_i is chosen uniformly at random between 1000 and 10000.
- the duration of compute operations a_i is chosen uniformly at random in interval [10,100] (time units).
- case (a): Application I/O load (b_i/a_i, and ultimately b_i) is chosen:
  1. Following a normal distribution of mean $\mu_1 = 0.1$, variance $\sigma_1 = 0.1$ truncated on the interval $[0, 1]$ for applications of low I/O load, the resulting mean being 0.289;
  2. Following a normal distribution of mean $\mu_2 = 0.9$, variance $\sigma_2 = 0.1$ truncated on the interval $[0, 1]$ for applications of high I/O load. The resulting mean being 0.711.
- case (b): Alternatively, we use a single application profile and b_i is chosen following a normal distribution of mean $\mu = \alpha$, variance $\sigma = 0.1$ truncated on the interval $[0, 1]$
- Finally, to instantiate the number of processors Q_i of each application, we use a distribution in the discrete set $\{2^j\}_{j=0...11}$ of mean $\bar{Q}$. This is obtained by replacing in Equation (III.5) all values by their average value as a first approximation to generate the workload.

$$\bar{Q} = \frac{P(\beta\mu_1 + (1-\beta)\mu_2)}{\alpha_{\text{gen}}(1 + \beta\mu_1 + (1-\beta)\mu_2)}.$$

In the following, we name these families of workloads, case (a): *binormal uniform* (BU) and case (b): *normal uniform* (NU).

IV.B.5 Evaluation criterion

In this work, we seek to design mapping strategies to optimize the usage of compute resources and I/O bandwidth. Hence, we need to solve a scheduling problem composed of two sub-problems at the same time.

First, we have to compute a *job schedule*, solution of the *mapping* problem which consists in choosing for each application the allocation of compute nodes during the execution timeframe of the batch.

Second, we have to compute an *I/O schedule*, which consists in deciding which application acquires the usage of the I/O node to access the PFS.

The relevance of such a schedule can be measured either from the platform administrator point of view or from the user point of view. It is important to notice that these two approaches, although not exactly opposites, can be in conflict.

Therefore, given a schedule, we use two different metrics to evaluate the relevance of our strategies according to these two angles.

1. The platform *utilization* of a schedule at any time is given as the ratio between the total work executed and the time. Typically, the utilization is an objective more *platform* oriented. The evolution of the platform utilization during the execution shows to what extend the resources are used.

2. The *stretch* ρ_i of an application J_i is the ratio between its minimal execution time and its actual execution time. A stretch of 1 means that the application is not impacted by the other applications running on the system. A stretch of 2 means that due to I/O contention, the application takes twice as long to execute as it would normally. Typically, the stretch is a *user* oriented objective.

To sum up, our general optimization problem is the following: given a batch of jobs running on a platform with an available I/O bandwidth to the PFS, and connected to P compute nodes. Find a schedule that either maximizes the total utilization, or that minimizes the maximum stretch, respecting the resource constraint (available number of nodes and no I/O bandwidth sharing). This reduces to finding the right allocation of I/O for the different applications.

IV.B.6 Relevance of static workloads

For the sake of simplicity, we chose to use static workloads where the amount of work to perform is defined beforehand. However, in real life, applications continue to arrive all the time. Both our model and such dynamic workloads are comparable when in a steady-state but ours presents a startup and a closure time that are not realistic. Nonetheless, the core of mapping and scheduling applications is mostly relevant when the system is under heavy load, typically during week days. A system administrator could choose to use times with low utilization, e.g. nights and week-end, to wait the completion of all applications submitted, before accepting a new one in order to avoid starvation. Alternatively, we can imagine an overlap of several schedules when the platform is not stressed ie starting a new schedule with the available resources as soon as the previous one exits steady-state. In our experiments, we have to confined ourselves to these possible scenarios.

IV.C Evaluation and Results

In this section, we present experimental evaluations of mappings using the pack or list-scheduling strategies. We start with an analysis of the algorithms' performance on the different workloads. Then, we discuss these data in the light of resource usage throughout the execution.

IV.C.1 Difference between workloads

Parameter dependencies We expect the evaluation results to depend heavily on workloads. As a preamble for experiment, let us go through their characteristic differences.

Mira-based workloads exhibit a dependency between job duration and the number of processors used. Considering that the amount of I/O operations they perform is pondered by the processors, it follows that there is a correlation linking all parameters of the application definition.

Compared to the uniform generation and due to the dependencies between different parameters, these workloads have more diverse applications profiles with more defined features. In the following, we see whether it has a positive or a negative impact on scheduling policies.

Noise filtering The random factor, intrinsic to the generation leads to some distorted workloads, with pathologically high I/O. For example, in the binomial workload case, the I/O and the processor are not independently chosen in order to fit the target I/O load. Consecutive imbalance in the first application defined can end up designing an abnormally loaded workload. To avoid these artifacts, we exclude such data from the generation.

Utilization We study the four heuristics defined in section IV.A (LS and the three packs variants) behavior. Utilization measurement throughout the execution for each workload profile are presented in Figures IV.3, IV.4, IV.5 and IV.6. The experiments have been performed for different I/O load values ranging from 0.2 to 0.8. Each trace presents the average on ten executions. Studies on single runs have been made beforehand to ensure representativeness.

For all compute-intensive workloads, LS scheduling makes a better use of the platform throughout the execution. However, LS utilization is degrading when the I/O load increases, especially when dealing with uniform workloads. In all cases, the platform utilization of LS is stable throughout the execution, fitting the constant constraint. The small under-utilization at the beginning of some schedules is explained by the few critical applications both prioritized and heavily-constrained.

As for pack-based scheduling, we can observe in most cases a startup time with high machine utilization at the beginning and a quick deterioration at the end when the workload is exhausted. During the steady-state, pack scheduling based on application length can achieve a comparable utilization as LS, and even outperform it when the I/O load increases. Packs based on different order is always worse than this one set apart for I/O intensive uniform workloads.

It means that for a dynamic workload which ensures constant application disposability for the scheduler, Pack scheduling may achieve a better platform utilization. However as it depends on the application sorting prior to pack building, it may also lead to starvation for low-priority applications.

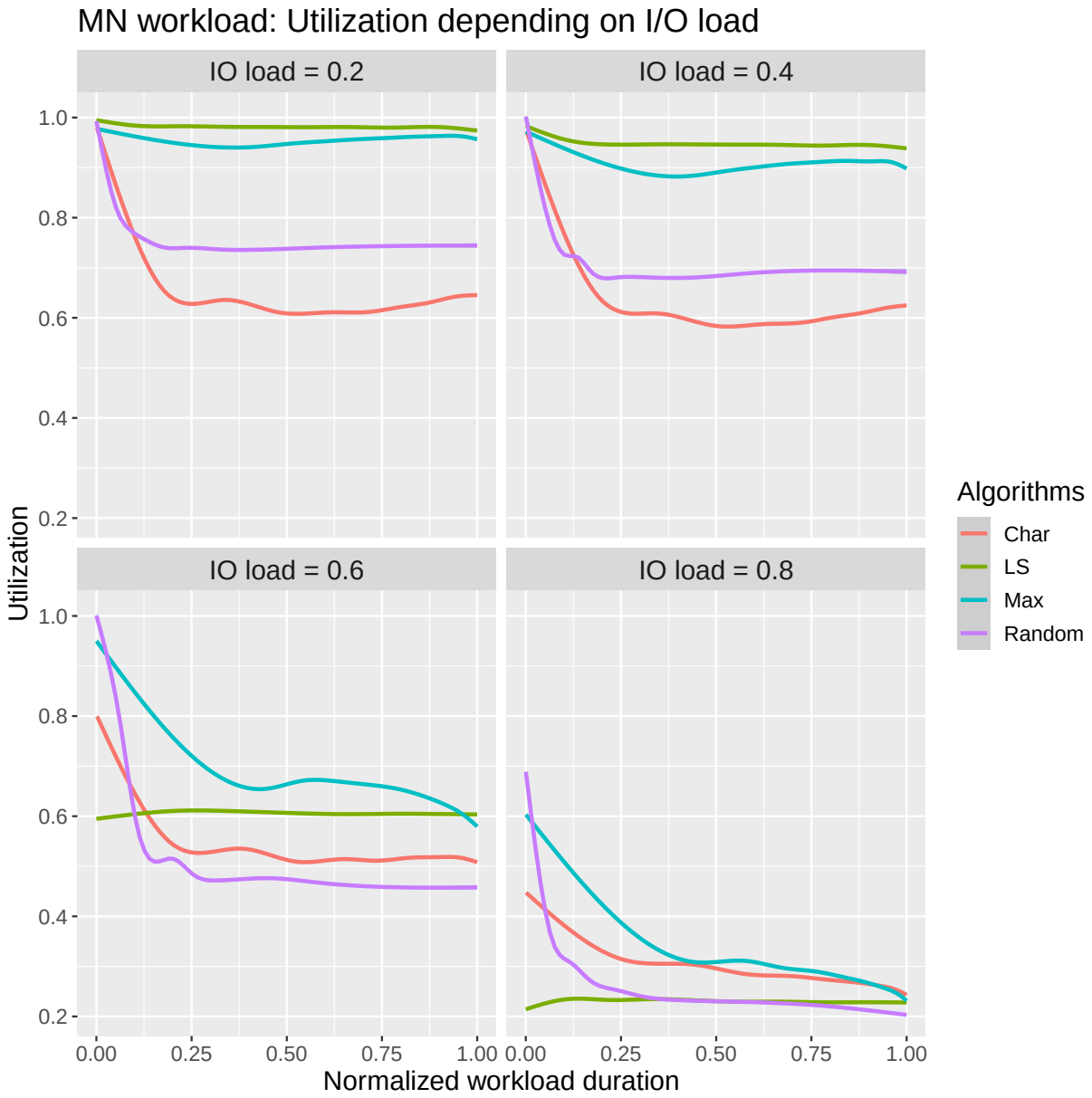


Figure IV.3: Utilization during the execution for different values of I/O load, with MN workload profile

Lessons learnt: List-scheduling performs consistently throughout the execution no matter the remaining available application. It can be outperformed by Pack-scheduling during the steady-state for high I/O loads (over 0.6).

Stretch Measurement on stretch as function of the I/O load are presented in Figure IV.7 for the maximum stretch and Figure IV.8 for the average stretch. Figures are

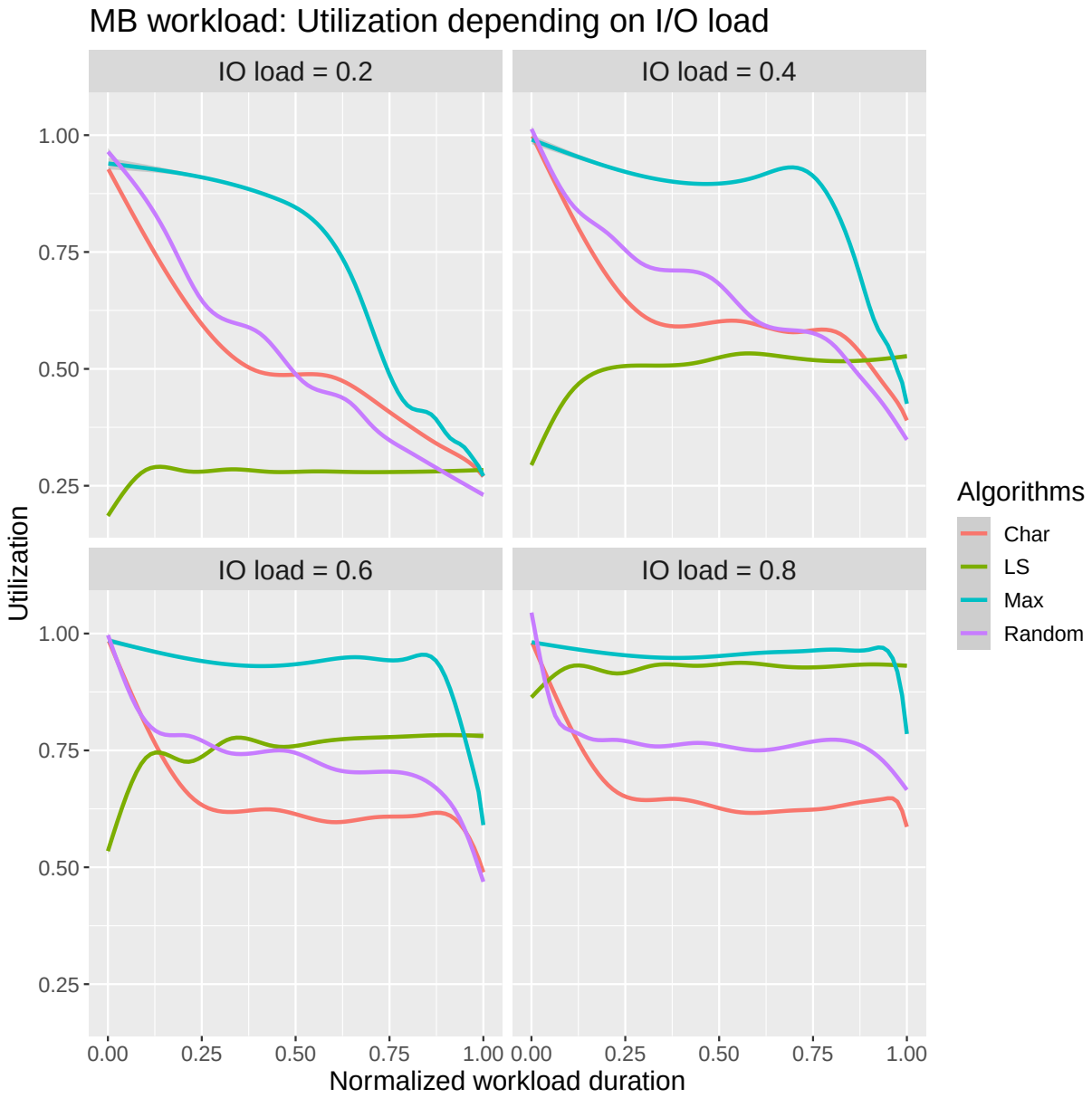


Figure IV.4: Utilization during the execution for different values of I/O load, with MB workload profile

divided depending on the underlying workload. In each trace, every point plots the result of one run on a specific workload. Lines show the average stretch with confidence intervals.

In all scenarios, pack scheduling provides a slight improvement on both the maximum and the mean stretch compared to LS.

Surprisingly packs built without sorting the applications have a lower average stretch on Mira based workloads. Indeed, the correlation between parameters leads

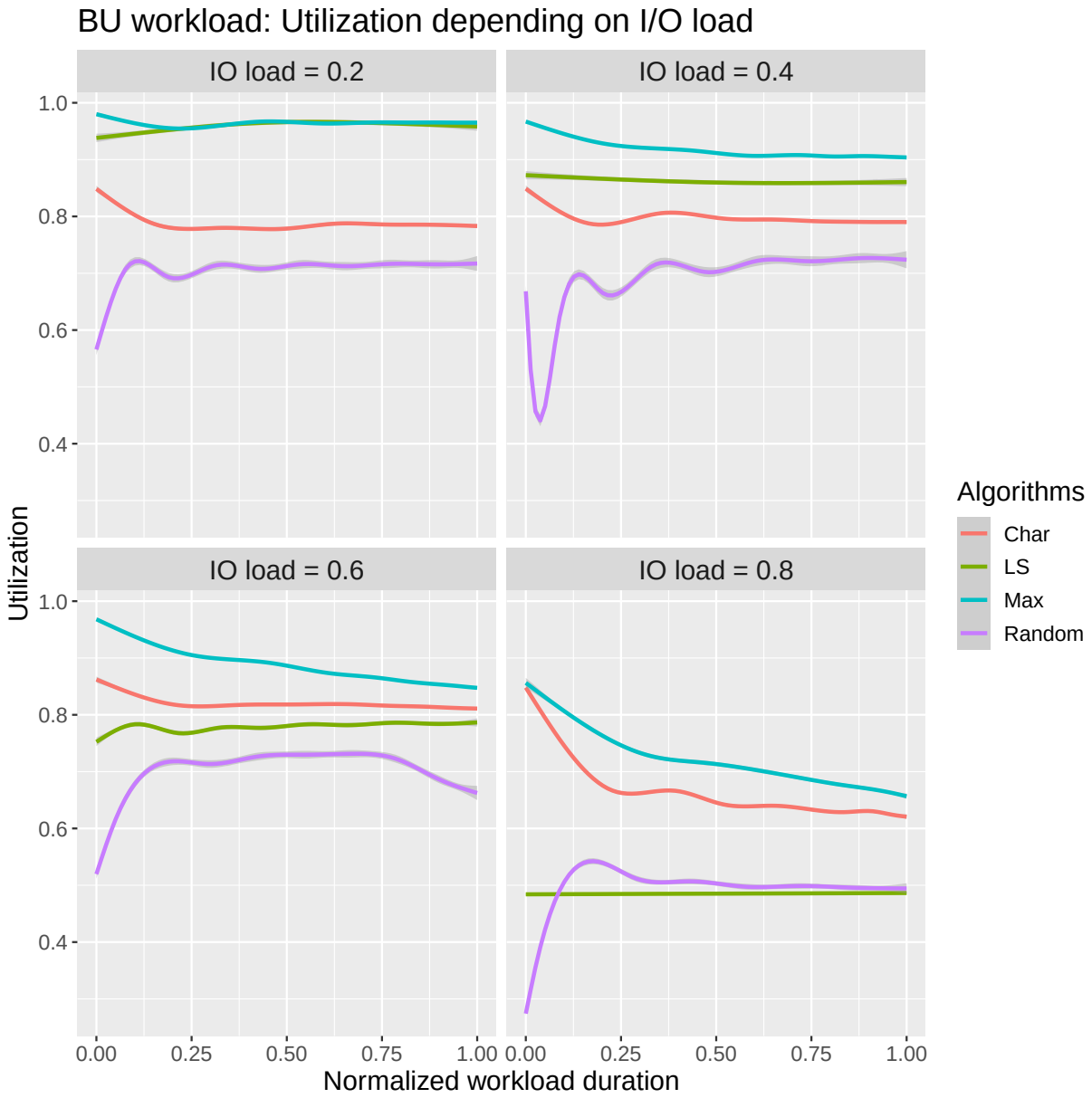


Figure IV.5: Utilization during the execution for different values of I/O load, with BU workload profile

to contention when sorting application based on the execution times.

Pack scheduling based on characteristic time was designed in order to synchronize applications and minimize the stretch. Indeed, it provides a significant improvement for uniform workloads and performs in the same way as other for the MN scenario.

The same heuristic performs worse than any other algorithm for MB (Mira profile and bi-distributed IO). Bi-distribution creates applications with the same characteristic time that are very different in their I/O behavior and the synchronization is made at

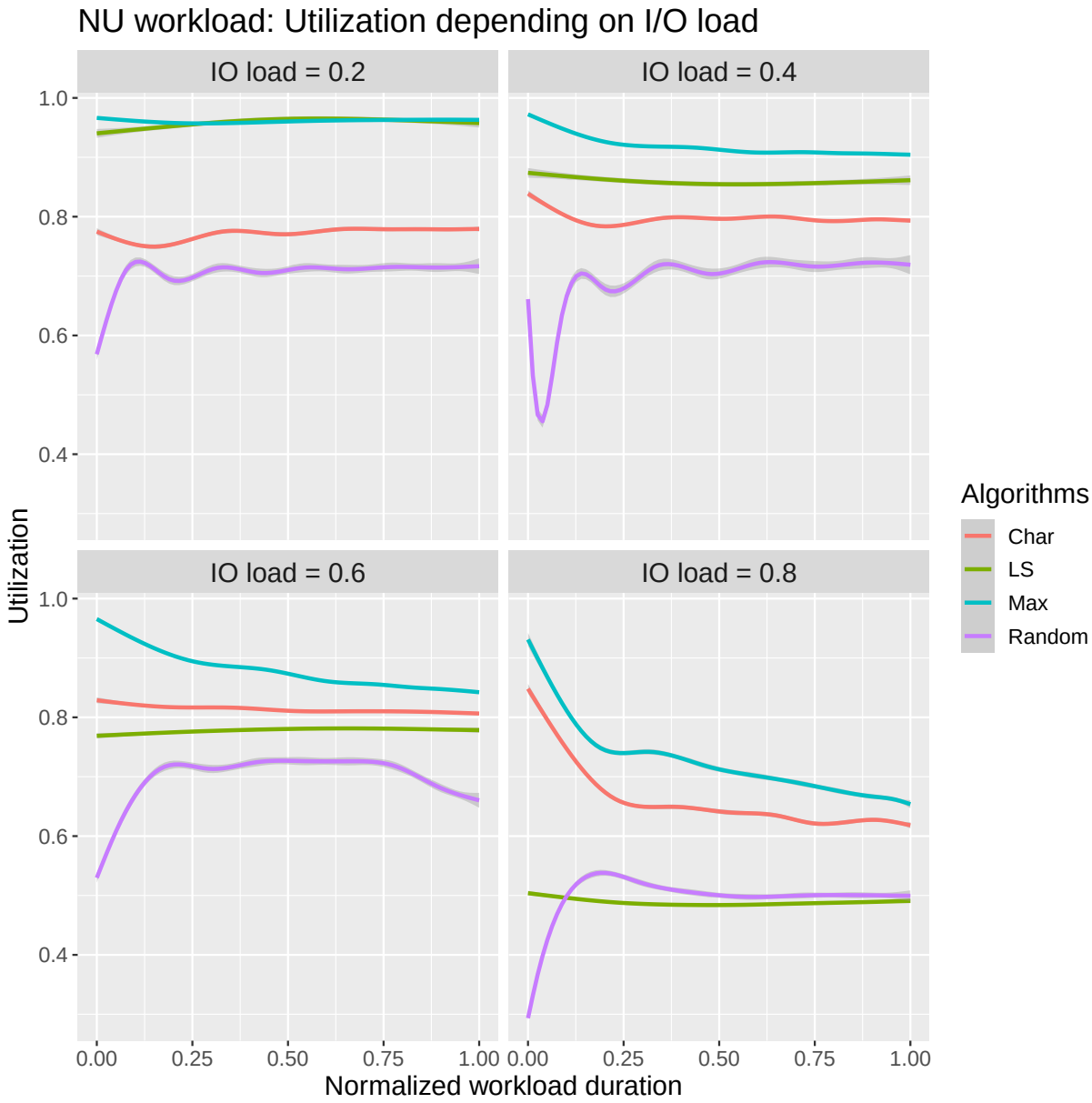


Figure IV.6: Utilization during the execution for different values of I/O load, with NU workload profile

loss. Indeed, when it happens, synchronism induces a constant delay for all concurrent applications. This may ensure a better bound for the worse case scenario by dividing equally the loss. However, when this delay is too long, it impacts all application executions.

One pitfall would be to evaluate pack scheduling with uniform workloads before implementing them. It could result in an overestimation of their performance. This case also emphasizes the setup sensibility and the difficulty to design solutions taking into

account both the general case and the specific.

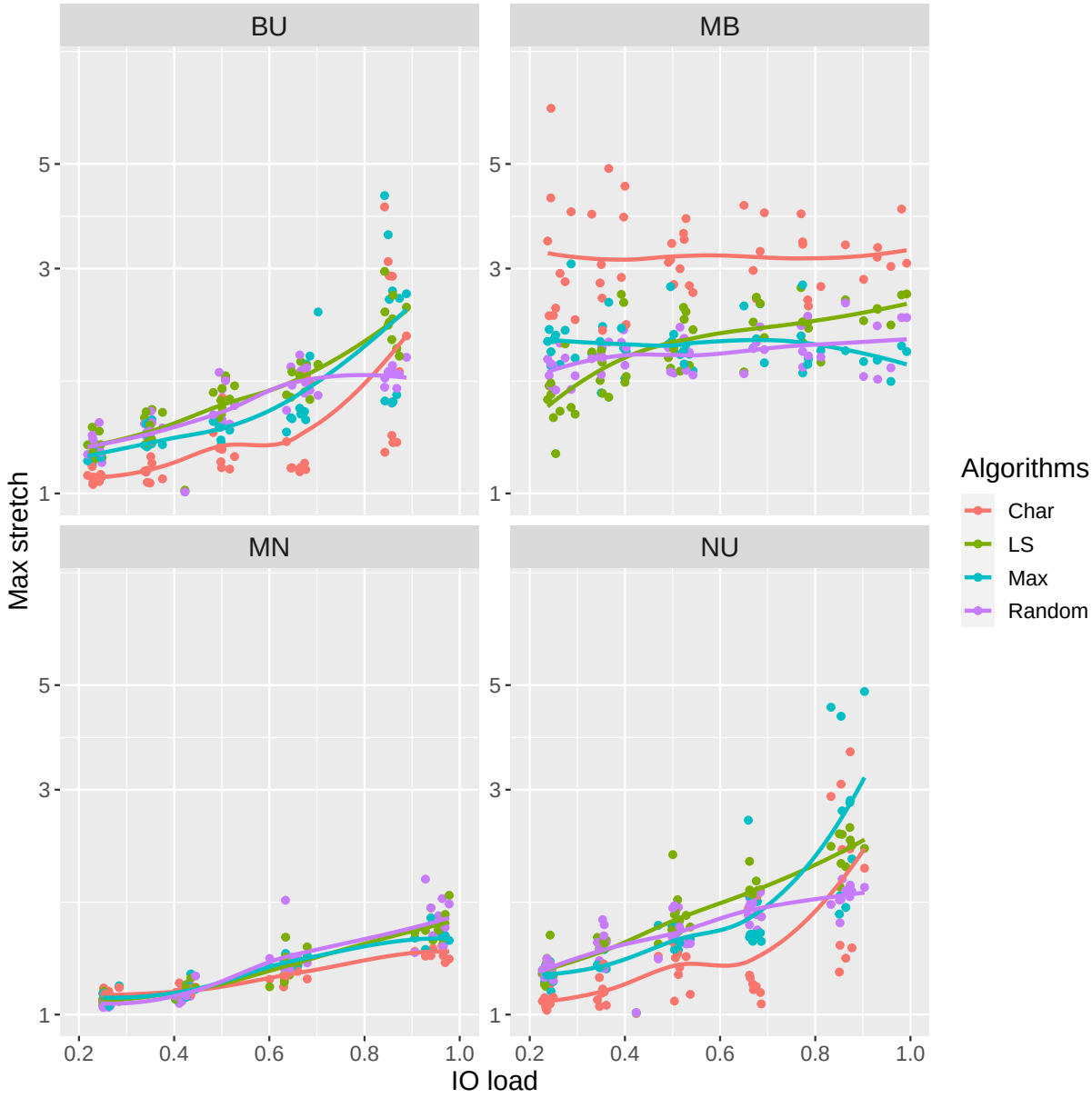


Figure IV.7: Max stretch for different workloads

IV.C.2 Difference in execution

In Figure IV.9, we present three measurement: the average number of applications running, the average number of processors used and the average portion of used bandwidth throughout the execution of each heuristic. The top three charts are obtained

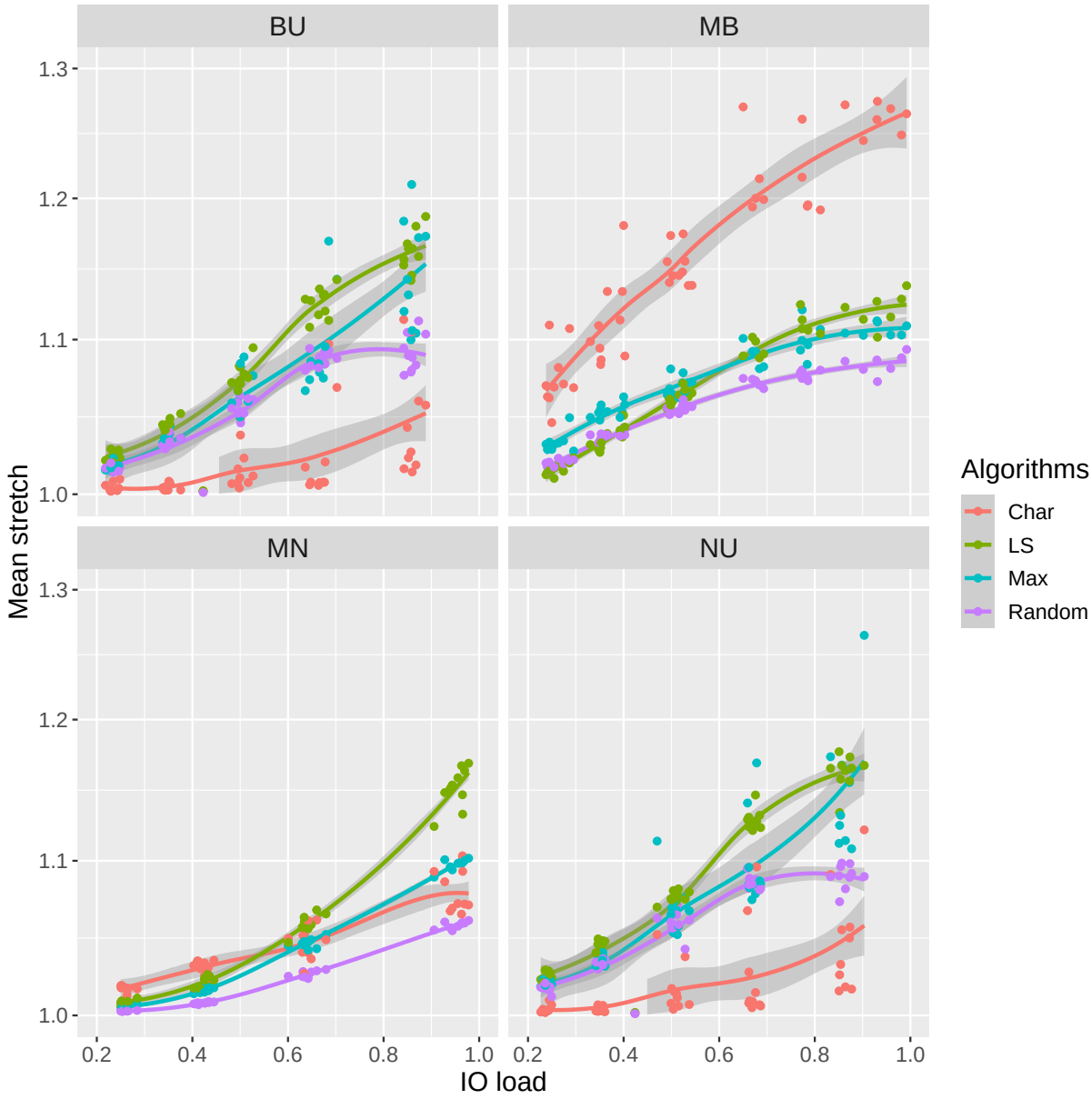


Figure IV.8: Mean stretch for different workloads

when scheduling applications from Mira Binormal workloads and the bottom three for Mira Normal Workloads. Comparing the two, we want to explain why Pack Scheduling is less efficient in the Mira Binormal case. We see that some applications profile are favored throughout the execution. It can be divided into phases.

1. The application that use more processors are scheduled first.

2. I/O intensive applications are performed last.

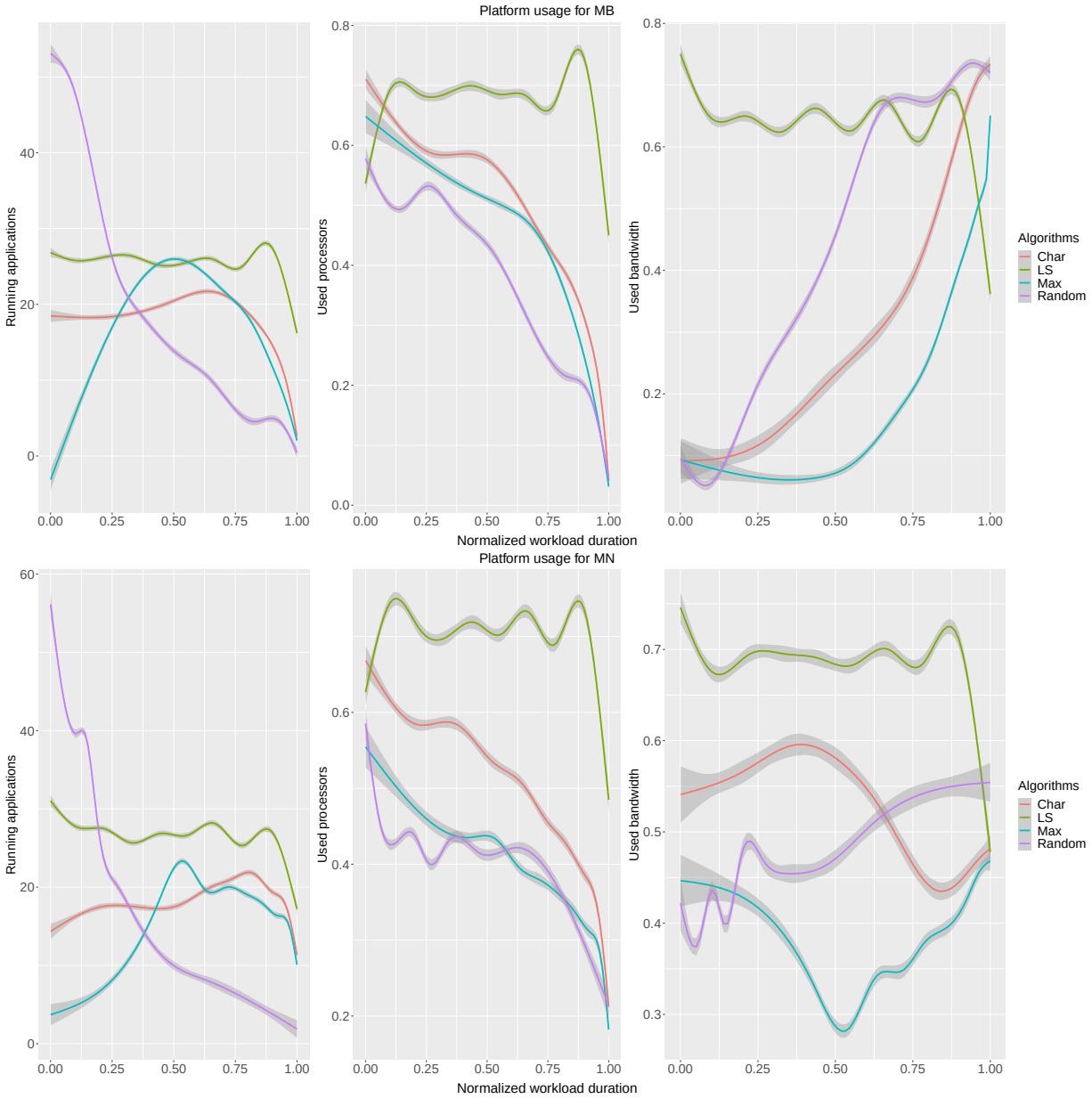


Figure IV.9: Resource usage (average number of applications running, used processors, used bandwidth on MB and MN workloads

Lessons learnt: Heuristics without steady-state are symptomatic of starvation (e.g. executions have several phases with I/O intensive applications performed last). Hence, when choosing an heuristic, it is crucial to check whether it reaches a steady-state or not. For instance, in Figure IV.9 no pack scheduling heuristic has a steady-state for MB workloads hence LS should be favored.

Idle time vs stretch

Depending of the constraint nature, the heuristics either choose to artificially add some idle time in order to avoid loosing time to contention (maximum or random pack strategies) or, in contrast increase the risk of conflict for a better resource usage (LS). More than a straight-up comparison between heuristic performance, it is this trade-off we discuss here.

In Figure IV.10, we show the idle ratio ie the mean stretch as a function of the accumulated processor-time of non-allocated resources over the total processor-time of the execution. The average of all experiments is drawn with confidence intervals in addition to a point for the result of each execution. Color gradient shows the final average occupation achieved during the workload execution. This figure was obtained while running with MB workloads. Similar results are obtained with MN whereas tendencies are harder to read in uniform cases.

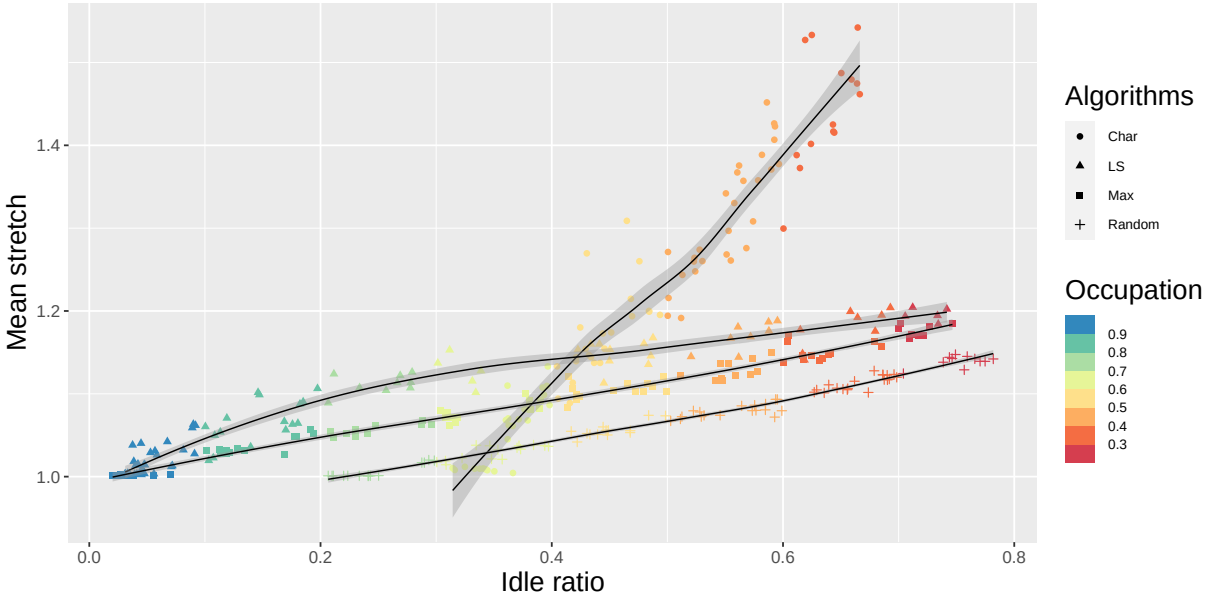


Figure IV.10: Mean stretch as a function of idle time

We see that standard pack and LS have a comparable Idle/stretch ratio with a slight offset in diminishing idle ratio and increasing stretch for list-scheduling. This offset gets in place for efficient workload (less than 10% idle time, and almost no stretch) and stays constant thereafter. Optimized pack built after sorting the applications by duration have exactly the same behavior as arbitrary constructed packs with only a constant gain in regard to idle time. It is more complex to interpret the behavior of packs based on characteristic time. It appears to introduce more idle time than an arbitrarily pack building policy in order to gain on the contention. However this only works for workloads that are already efficient. When the loss increases, the stretch skyrockets. It is likely due to the fact that applications synchronize their I/O, therefore setting a

constant delay for all applications running together. For compute intensive workloads and for sets of comparable applications, this delay is close to zero hence the gain, but when workloads become data intensive all applications are penalized.

Lesson learnt: When I/O increases and conflicts occur, both the idle time and the contention increase. Packs based on characteristic time are advantageous under heavy I/O loads and must be chosen only after rigorous study. As for other heuristics, they keep the same trade-off pace. Then, choosing one over another depends only on the initial offset preference; in addition to the performance discussed before.

IV.D Conclusion and prospects

In this chapter, we presented strategies for data-aware mapping. They were chosen for their simplicity and the clear difference on the constraints they implement. The analysis of LS enables to study strict constraints whereas Pack scheduling represents average restrictions. We defined evaluation metrics and designed different workloads in order to provide a deep understanding of the strategies' behavior. Results show that Pack scheduling can provide a better machine utilization throughout the execution when the I/O exertion is high, in other cases LS is preferable. The parameters defining the workload are also capital with a trade-off between optimized stretch and better platform utilization. For this purpose, we implemented pack based on characteristic time and showed that, if the I/O exertion is not excessive, it can achieve better fairness than both LS and other packs in the same scenarios. However, we have seen that performance depends strongly on the workload, pack-scheduling algorithms, in particular the one using characteristic time, perform better with comparable applications (uniform generation, normal repartition of I/O). The natural direction to open-up as continuation is to discuss ways to classify applications in order to be more efficient in pack building and have a finer workload appreciation. In a more elaborate way and more importantly, we plan to allow parallel I/O operations instead of exclusive ones. In such a set-up, we would be able to study not only single mapping strategies but also the combination of several strategies, sharing an adequately dedicated portion of the bandwidth.

Chapter V

Conclusion

HPC applications grew more and more data intensive with the advent of high-performance data analytic and deep learning. In this context, the strain exerted on the shared bandwidth of a platform became a concern. The current thesis is a part of the recent developments aiming to include I/O awareness in the design of theoretical models, and mapping and scheduling strategies.

V.A Retrospective

Early work In Chapter II, we assumes a upstream allocation of the machine and focus on the restricted field of I/O accesses for applications that alternate computation and I/O. This work mainly serves as a basis for further development by describing a simple model taking into account the I/O pattern of HPC applications. The simple model serves to set the first stone to discuss the trade-off between platform performance and user fairness. Indeed, we proved that the problem is NP-complete and provide strategies for simple cases. However, it gets harder to prove interesting result with more exotic metrics. More interestingly, there is a gap between the theoretical approximation result, here "all list-scheduling heuristics provide a 2-approximation no matter the order" and the practical result where the order has an obvious impact on the platform usage. A theoretical analysis is necessary to enrich and discuss further the heuristics. Reciprocally, there is little hope to gain understanding on application behavior by simulation without a proper definition of the parameters, of the problem and the metrics to observe, in other words, without the theoretical development whose deepness conditions our insight on the subject. This early work, although imperfect and improvable, contributes to sketch the approach we refined these last years. That is to say a come and go between mathematical modeling and experimental measurement using simulation.

Second step, a change of scope At this point, we had the different directions to continue, the first, unexplored one was to add complexity on the model: tackling non-periodic applications or including malleability for example. It seemed unlikely to pro-

vide strong and relevant theoretical model for this problems. Instead, we chose to broaden the scope and ask the question of the application mapping on the platform. The subject connects: what is malleability if not a variation of the allocated resources? In this case, mapping the application is a prerequisite for any further development. We discuss this issue in Chapter III. Addressing application mapping represents a change of scale on the approach. We moved from a point of view considering only applications to one that encompasses the platform. To this purpose, we designed a platform model, we also had to define a better characterization of the I/O intensity of a workload to take into account the overall behavior. Based on this model, we propose a simple I/O aware mapping strategy and evaluated it with simulation. The results were surprisingly good at showing the trade-off between control (and fairness) and overall performance. It strengthens my personal conviction that there is no absolute "best strategy" for mapping and scheduling applications. It is up to system administrators to choose in what extend they want to exercise control on their platform depending on their way to "recover" sacrificed resources. One exciting and contemporary direction arising from this consideration is the link to make with energy management. Sacrificing (a little) platform performance for a better resource usage is a direction to dig in the process to make more responsible supercomputers.

Despite our interest in the matter, we could not build a connection on this two field strong enough to justify investing a part of our already limited allocated time in this area.

Another very stimulating thing in this second part was the collaboration with the ARCOS team members in Madrid. Indeed, my skills tends to an empirical approach of computer science, they provided their own software to handles I/O operations. We were allowed to implement my policy in a dedicated part of the code. Thanks to them, we were able to connect my first works to a functional software.

Maturation in the approach The encouraging result of this part opened up new questions. First of all, pack scheduling enforces strict restrictions on application execution. Is it really pertinent to use compared to another less rigid I/O-aware strategy? Secondly, can we measure whether a schedule is good for waste-control (backfilling, power-saving...) or not? Attempts in the domain were not very convincing: they implied to design time-dependent metrics which could not only render the platform occupation but its repartition during the execution. This trial and error contributes to build more polished metrics in the following chapter. The Chapter IV answer the first question by building a comparison with List-scheduling. However it revealed itself to be more interesting than the simple opposition of algorithms. The two strategies translate different resource usage philosophy and might both be interesting depending on the objective or the set-up. One more discrete evolution in our approach was the return to more realistic workload generation. Indeed the scientific material took long enough to mature. If what we wanted to do was straightforward, the process was slowed down not only by the degraded work condition but also by the result variability. We found out that

the performance of a strategy depended on characteristic of the underlying workload. In this condition, we seek to extract relevant indicators and design application accordingly. Finally, we made the choice to compare our wide-range generated workloads to one inspired by the literature. The first approach making independent parameters definition thus exploring a patternless wide-range of parameters while the second one linked parameters one to the other.

V.B First thoughts on an open problem

When we compared Pack scheduling to List scheduling, we underlined the importance of application similarity to choose which strategy is the best. We know that, in a real-life systems, we can find different groups of similar applications. We make the following observations: Exclusive access is penalizing applications with small periods when they have to wait for the I/O completion of a long application. Conversely, when applications with the same periods are sharing the bandwidth they suffer a slowdown at each iteration.

As a closing point for this manuscript, we will provide some discussion on this ongoing research.

V.B.1 Openings

Previously, we developed the intuition that under certain conditions applications applications benefits from scheduling policies in exclusive access. In other cases, best effort perform well. We illustrate these possibilities in Figs V.1 and V.2. In these figures, two applications perform alternatively computation and I/O phases following either "exclusive access" or "fair share" policies. In Figure V.1, applications have significantly different characteristic time. With these parameters, "fair share" is more efficient than "exclusive access". Indeed, the delay for Application 2 can be consequent as regard to its period time.

In comparison, Figure V.2 shows similar applications. In this example, after the first period, applications are synchronized and there is no more delay.

We extend this problematic by considering subsets of applications performing exclusive access in their dedicated part of the bandwidth. The problem becomes: given a set of applications on a platform with finite bandwidth, what is the best partitioning of application into exclusive subsets and what is the optimal bandwidth for each part?

V.B.2 Models

We consider a set of applications $\mathcal{A}$ defined solely by two parameters.

1. a characteristic time t_i
2. an amount of I/O operations to perform in this time.

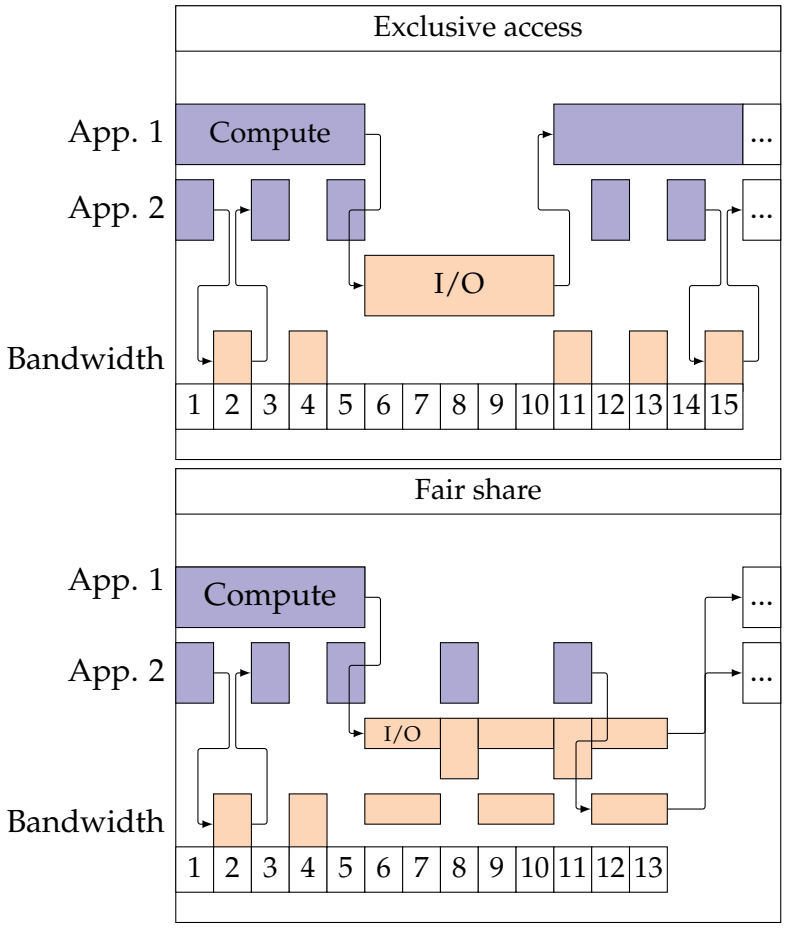


Figure V.1: Example with dissimilar applications

Define a partition of the applications a set $(I_k)_{n\in\mathbb{N}} \subset \mathcal{A}^n$ such that $\forall k$ A partitioning of the bandwidth has to follow two constraints. First, the sum of the part bandwidths B_I cannot exceed the total bandwidth of the platform. Secondly, there is a lower bound to the bandwidth dedicated to a part in order to perform all I/O operations within the allocated time.

1. $\sum B_i \leqslant B$
2. $\sum \frac{v_i}{t_i} \leqslant B_i$

The objective is to minimize a simplified version of the stretch. This metric is the maximum ratio between the longest execution time in each pack ($\frac{max(v_i)}{B_i}$) and the shortest one in optimal conditions ($\frac{min(v_i)}{B}$)

$$max\left(\frac{max(v_i)}{min(v_i)} \cdot \frac{B}{B_i}\right) \tag{V.1}$$

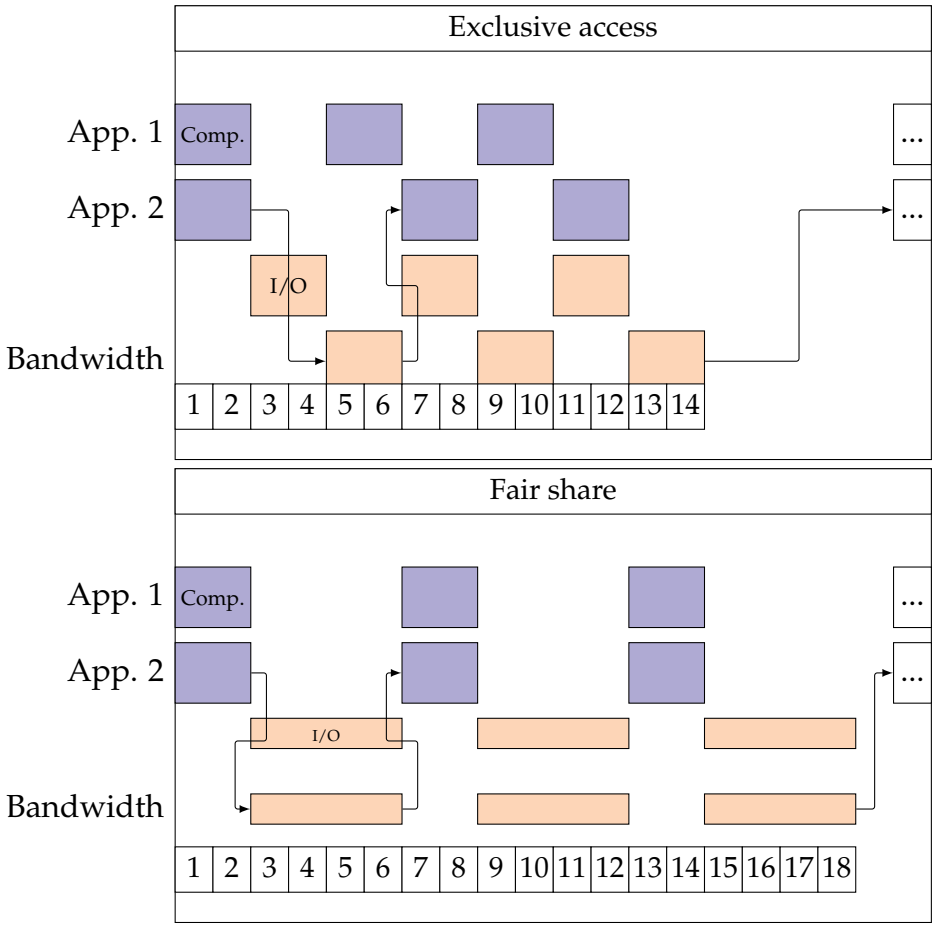


Figure V.2: Example with similar applications

V.B.3 Examples

The general idea, in this model, is that regrouping applications improves the group bandwidth (B_i, increasing the I/O throughput) while degrading the stretch (ratio $\frac{max(v_i)}{min(v_i)}$, delay before starting an I/O operation).

Putting applications with similar v_i only increases slightly the ratio $\frac{max(v_i)}{min(v_i)}$ while decreasing $\frac{B}{B_i}$. Therefore an optimal partition would be based on the similarity of v_i (Intuition 1).

Preliminary experiments testing all partitioning of a small number of applications show this property. However, in our model, an application with an important $\frac{v_i}{t_i}$ ratio has an impact on the bandwidth allocated to its group. Therefore, it may be optimal to place such an application in a group with critical bandwidth regardless of other parameters. We aim to define "reasonable constraints" preventing this scenario.

Intuition 1. "Under reasonable constraints" $\exists (I_n)_n \; optimal \; s.t \; \forall n, \; i,j \in I_n \Rightarrow \forall v_k \in [v_i, v_j], k \in I_n$

As a preliminary evaluation, we simulated small sets of applications and tested all possible partitioning of the applications. Showing that, in any case, there is always an optimal partition that respect v_i continuity. In most cases, there is an improvement to build subsets of applications compared to the all-shared or all exclusive I/O accesses. An example is presented in Figure V.3, with color showing different application group, and the axis the parameters. Group are continuous in regards to v_i.

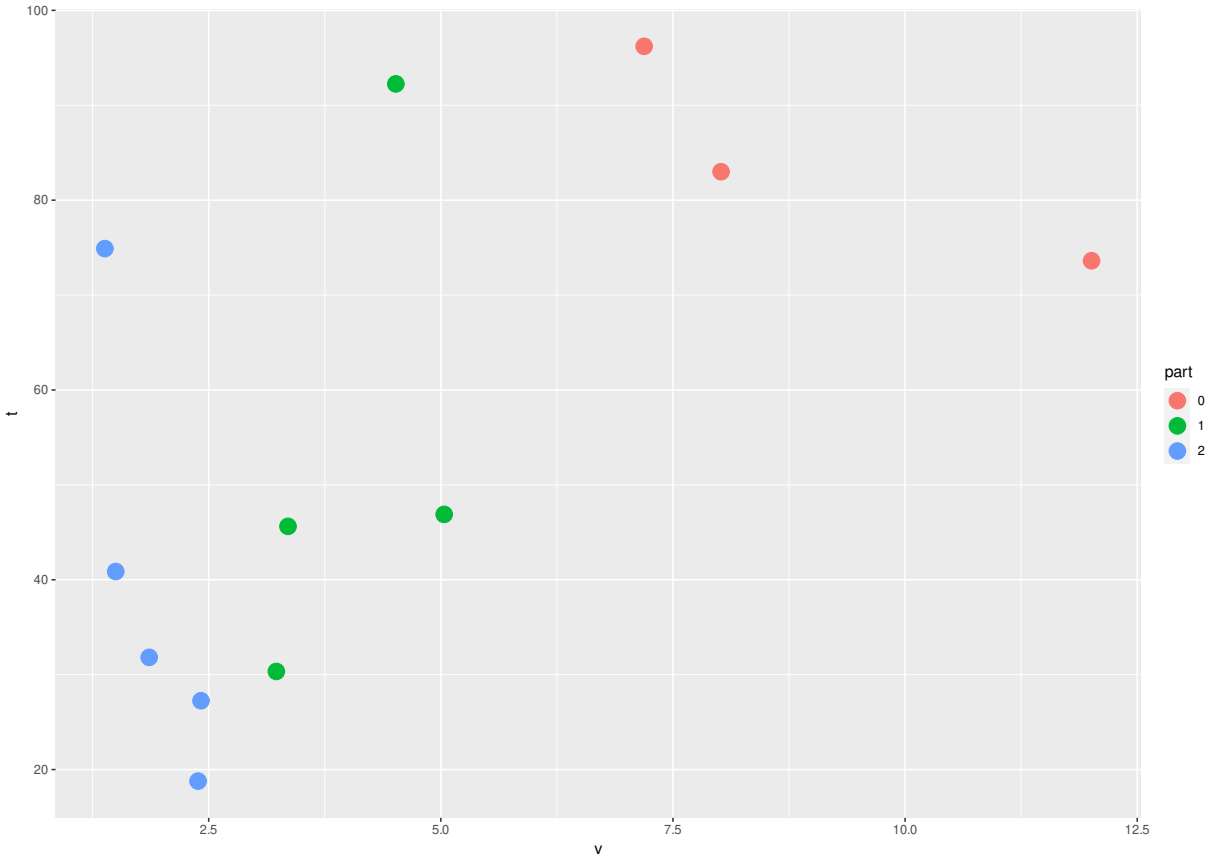


Figure V.3: Example of an optimal application partitioning

Conclusion We are currently working on how to define groups of applications in a realistic set-up with the help of the SimGrid [17] framework for experimental studies.

References

[1] Computer Architecture and Technology Area. https://www.arcos.inf.uc3m.es/. Accessed: 2020-10-6.

[2] PlaFRIM cluster. https://www.plafrim.fr/. Accessed: 2020-07-27.

[3] Slurm Multifactor Priority Plugin. https://slurm.schedmd.com/priority_multifactor.html. Accessed: 2020-10-06.

[4] Specifications of Aurora exascale machine. https://aurora.alcf.anl.gov/. Accessed: 2020-07-01.

[5] Supercomputer Fugaku development. https://www.fujitsu.com/global/about/innovation/fugaku/documents/fugaku-brochure-en.pdf. Accessed: 2021-07-01.

[6] Maicon Melo ALVES et Lucia Maria de ASSUMPÇAO DRUMMOND : A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software*, 128:150 – 163, 2017.

[7] Guillaume AUPY, Olivier BEAUMONT et Lionel EYRAUD-DUBOIS : What size should your buffers to disks be? *In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 660–669. IEEE, 2018.

[8] Guillaume AUPY, Olivier BEAUMONT et Lionel EYRAUD-DUBOIS : Sizing and partitioning strategies for burst-buffers to reduce io contention. *In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 631–640. IEEE, 2019.

[9] Guillaume AUPY, Ana GAINARU et Valentin LE FÈVRE : Periodic i/o scheduling for super-computers. *In International workshop on performance modeling, benchmarking and simulation of high performance computer systems*, pages 44–66. Springer, 2017.

[10] Guillaume AUPY, Manu SHANTHARAM, Anne BENOIT, Yves ROBERT et Padma RAGHAVAN : Co-scheduling algorithms for high-throughput workload execution. *Journal of Scheduling*, 19(6):627–640, 2016.

[11] Babak BEHZAD, Huong Vu Thanh LUU, Joseph HUCHETTE, Surendra BYNA, Ruth AYDT, Quincey KOZIOL, Marc SNIR *et al.* : Taming parallel i/o complexity with auto-tuning. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.

[12] J. L. BEZ, F. Z. BOITO, L. M. SCHNORR, P. O. A. NAVAUX et J. MÉHAUT : Twins: Server access coordination in the I/O forwarding layer. *In 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 116–123, March 2017.

[13] Raphaël BLEUSE, Konstantinos DOGEAS, Giorgio LUCARELLI, Grégory MOUNIÉ et Denis TRYSTRAM : Interference-aware scheduling using geometric constraints. *In European Conference on Parallel Processing*, pages 205–217. Springer, 2018.

[14] Francieli Zanon BOITO, Rodrigo Virote KASSICK, Philippe OA NAVAUX et Yves DENNEULIN : Agios: Application-guided i/o scheduling for parallel file systems. *In 2013 International Conference on Parallel and Distributed Systems*, pages 43–50. IEEE, 2013.

[15] Peter BRUCKER et P BRUCKER : *Scheduling algorithms*, volume 3. Springer, 2007.

[16] Philip CARNS, Robert LATHAM, Robert ROSS, Kamil ISKRA, Samuel LANG et Katherine RILEY : 24/7 characterization of petascale i/o workloads. *In 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[17] Henri CASANOVA, Arnaud GIERSCH, Arnaud LEGRAND, Martin QUINSON et Frédéric SUTER : Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, juin 2014.

[18] Zhendong CHENG, Zhongzhi LUAN, You MENG, Yijing XU, Depei QIAN, Alain ROY, Ning ZHANG et Gang GUAN : Erms: An elastic replication management system for HDFS. *In Cluster Computing Workshops, 2012 IEEE International Conference on*, pages 32–40. IEEE, 2012.

[19] J. T. DALY : A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3):303–312, 2006.

[20] Sheng DI et Franck CAPPELLO : Fast error-bounded lossy hpc data compression with sz. *In 2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[21] M. DORIER, S. IBRAHIM, G. ANTONIU et R. ROSS : Using formal grammars to predict I/O behaviors in HPC: The omnisc'io approach. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2435–2449, Aug 2016.

[22] 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 2012 IEEE International Conference on Cluster Computing*, pages 155–163. IEEE, 2012.

[23] Matthieu DORIER, Gabriel ANTONIU, Rob ROSS, Dries KIMPE et Shadi IBRAHIM : Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. *In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 155–164. IEEE, 2014.

[24] Matthieu DORIER, Shadi IBRAHIM, Gabriel ANTONIU et Rob ROSS : Omnisc'io: a grammar-based approach to spatial and temporal i/o patterns prediction. *In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 623–634. IEEE, 2014.

[25] Matthieu DREHER et Bruno RAFFIN : A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. *In Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 277–286. IEEE, 2014.

[26] Dror G FEITELSON : *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[27] Eitan FRACHTENBERG et Dror G FEITELSON : Pitfalls in parallel job scheduling evaluation. *In Workshop on Job Scheduling Strategies for Parallel Processing*, pages 257–282. Springer, 2005.

[28] Eitan FRACHTENBERG, G FEITELSON, Fabrizio PETRINI et Juan FERNANDEZ : Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed systems*, 16(11):1066–1077, 2005.

[29] Ana GAINARU, Guillaume AUPY, Anne BENOIT, Franck CAPPELLO, Yves ROBERT et Marc SNIR : Scheduling the i/o of hpc applications under congestion. *In 2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[30] Ana GAINARU, Valentin LE FÈVRE et Guillaume PALLEZ : I/o scheduling strategy for periodic applications. *ACM Transactions on Parallel Computing*, 2019.

[31] Markus GEIMER, Felix WOLF, Brian JN WYLIE, Erika ÁBRAHÁM, Daniel BECKER et Bernd MOHR : The scalasca performance toolset architecture. *Concurrency and computation: Practice and experience*, 22(6):702–719, 2010.

[32] Robert GRANDL, Ganesh ANANTHANARAYANAN, Srikanth KANDULA, Sriram RAO et Aditya AKELLA : Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, août 2014.

[33] Salman HABIB, Adrian POPE, Hal FINKEL, Nicholas FRONTIERE, Katrin HEITMANN, David DANIEL, Patricia FASEL, Vitali MOROZOV, George ZAGARIS, Tom PETERKA, Venkatram VISHWANATH, Zarija LUKIĆ, Saba SEHRISH et Wei keng LIAO : Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[34] Yuya HASHIMOTO et Kento AIDA : Evaluation of performance degradation in hpc applications with vm consolidation. *In Networking and Computing (ICNC), 2012 Third International Conference on*, pages 273–277. IEEE, 2012.

[35] Katrin HEITMANN, Nicholas FRONTIERE, Esteban RANGEL, Patricia LARSEN, Adrian POPE, Imran SULTAN, Thomas URAM, Salman HABIB, Hal FINKEL, Danila KORYTOV *et al.* : The last journey. i. an extreme-scale simulation on the mira supercomputer. *The Astrophysical Journal Supplement Series*, 252(2):19, 2021.

[36] Stephen HERBEIN, Dong H AHN, Don LIPARI, Thomas RW SCOGLAND, Marc STEARMAN, Mark GRONDONA, Jim GARLICK, Becky SPRINGMEYER et Michela TAUFER : Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. *In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80, 2016.

[37] Anthony JG HEY, Stewart TANSLEY, Kristin Michele TOLLE *et al.* : *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[38] Florin ISAILA, Francisco Javier Garcia BLAS, Jesus CARRETERO, Wei keng LIAO et Alok CHOUDHARY : A scalable message passing interface implementation of an ad-hoc parallel I/O system. *The International Journal of High Performance Computing Applications*, 24(2):164–184, 2010.

[39] Florin ISAILA, Jesus CARRETERO et Rob ROSS : Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms. *In Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 346–355. IEEE, 2016.

[40] Mihailo ISAKOV, Eliakin DEL ROSARIO, Sandeep MADIREDDY, Prasanna BALAPRAKASH, Philip CARNS, Robert B ROSS et Michel A KINSY : Hpc i/o throughput bottleneck analysis with explainable local models. *In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2020.

[41] David JACKSON, Quinn SNELL et Mark CLEMENT : Core algorithms of the maui scheduler. *In Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[42] Aw KHAN, Hyogi SIM, Sudharshan S VAZHKUDAI, Ali R BUTT et Youngjae KIM : An analysis of system balance and architectural trends based on top500 supercomputers. *In The International Conference on High Performance Computing in Asia-Pacific Region*, pages 11–22, 2021.

[43] Harsh KHETAWAT, Christopher ZIMMER, Frank MUELLER, Scott ATCHLEY, Sudharshan VAZHKUDAI et Misbah MUBARAK : Evaluating burst buffer placement in hpc systems. Rapport technique, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2019.

[44] Sunggon KIM, Alex SIM, Kesheng WU, Suren BYNA, Yongseok SON et Hyeonsang EOM : Towards hpc i/o performance prediction through large-scale log analysis. *In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 77–88, 2020.

[45] Anthony KOUGKAS, Matthieu DORIER, Rob LATHAM, Rob ROSS et Xian-He SUN : Leveraging burst buffer coordination to prevent i/o interference. *In e-Science (e-Science), 2016 IEEE 12th International Conference on*, pages 371–380. IEEE, 2016.

[46] Sidharth KUMAR, Avishek SAHA, Venkatram VISHWANATH, Philip CARNS, John A SCHMIDT, Giorgio SCORZELLI, Hemanth KOLLA, Ray GROUT, Robert LATHAM, Robert ROSS *et al.* : Characterization and modeling of pidx parallel i/o for performance optimization. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 67. ACM, 2013.

[47] J.K. LENSTRA, A.H.G. RINNOOY KAN et P. BRUCKER : Complexity of machine scheduling problems. *Ann. of Discrete Math.*, 1:343–362, 1977.

[48] Joseph LEUNG, Laurie KELLY et James H. ANDERSON : *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[49] Y. LI, X. LU, E. L. MILLER et D. D. E. LONG : Ascar: Automating contention management for high-performance storage systems. *In 2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–16, May 2015.

[50] Harold C LIM, Shivnath BABU et Jeffrey S CHASE : Automated control for elastic storage. *In Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[51] 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 Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[52] Y. LIU, R. GUNASEKARAN, X. MA et S. S. VAZHKUDAI : Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. *In SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829, Nov 2016.

[53] 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 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[54] Sandeep MADIREDDY, Prasanna BALAPRAKASH, Philip CARNS, Robert LATHAM, Robert ROSS, Shane SNYDER et Stefan WILD : Modeling i/o performance variability using conditional variational autoencoders. *In 2018 IEEE international conference on cluster computing (CLUSTER)*, pages 109–113. IEEE, 2018.

[55] Barton P MILLER, Mark D. CALLAGHAN, Jonathan M CARGILLE, Jeffrey K HOLLINGSWORTH, R Bruce IRVIN, Karen L KARAVANIC, Krishna KUNCHITHAPADAM et Tia NEWHALL : The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[56] James OLY et Daniel A REED : Markov model prediction of i/o requests for scientific applications. *In Proceedings of the 16th international conference on Supercomputing*, pages 147–155, 2002.

[57] Scott PARKER, Vitali MOROZOV, Sudheer CHUNDURI, Kevin HARMS, Chris KNIGHT et Kalyan KUMARAN : Early evaluation of the cray xc40 xeon phi system 'theta'at argonne. Rapport technique, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[58] Tirthak PATEL, Suren BYNA, Glenn K LOCKWOOD et Devesh TIWARI : Revisiting i/o behavior in large-scale storage systems: the expected and the unexpected. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[59] Tirthak PATEL, Zhengchun LIU, Raj KETTIMUTHU, Paul RICH, William ALLCOCK et Devesh TIWARI : Job characteristics on large-scale systems: long-term analysis, quantification, and implications. *In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

[60] Vincent PILLET, Jesús LABARTA, Toni CORTES et Sergi GIRONA : Paraver: A tool to visualize and analyze parallel code. *In Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. Citeseer, 1995.

[61] Manu SHANTHARAM, Youngtae YOUN et Padma RAGHAVAN : Speedup-aware co-schedules for efficient workload management. *Parallel Processing Letters*, 23(02): 1340001, 2013.

[62] Sameer S SHENDE et Allen D MALONY : The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[63] David E. SINGH et Jesus CARRETERO : Combining malleability and i/o control mechanisms to enhance the execution of multiple applications. *Journal of Systems and Software*, 148:21 – 36, 2019.

[64] David E SINGH, Florin ISAILA, Alejandro CALDERÓN, Felix GARCIA et Jesús CARRETERO : Multiple-phase collective i/o technique for improving data access locality. *In 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, pages 534–542. IEEE, 2007.

[65] 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.

[66] Shane SNYDER, Philip CARNS, Kevin HARMS, Robert ROSS, Glenn K LOCKWOOD et Nicholas J WRIGHT : Modular hpc i/o characterization with darshan. *In 2016 5th workshop on extreme-scale programming tools (ESPT)*, pages 9–17. IEEE, 2016.

[67] Shane SNYDER, Philip CARNS, Robert LATHAM, Misbah MUBARAK, Robert ROSS, Christopher CAROTHERS, Babak BEHZAD, Huong Vu Thanh LUU, Surendra BYNA et PRABHAT : Techniques for modeling large-scale HPC I/O workloads. *In Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[68] H. SUN, R. ELGHAZI, A. GAINARU, G. AUPY et P. RAGHAVAN : Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling. *In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 194–203, May 2018.

[69] V TANAEV, W GORDON et Yakov M SHAFRANSKY : *Scheduling theory. Single-stage systems*, volume 284. Springer Science & Business Media, 2012.

[70] 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, Sept 2017.

[71] François TESSIER, Preeti MALAKAR, Venkatram VISHWANATH, Emmanuel JEANNOT et Florin ISAILA : Topology-aware data aggregation for intensive i/o on large-scale supercomputers. *In 2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 73–81. IEEE, 2016.

[72] François TESSIER, Venkatram VISHWANATH et Emmanuel JEANNOT : Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers. *In 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 70–80. IEEE, 2017.

[73] Tony T. TRAN, Meghana PADMANABHAN, Peter Yun ZHANG, Heyse LI, Douglas G. DOWN et J. Christopher BECK : Multi-stage resource-aware scheduling for data centers with heterogeneous servers. *Journal of Scheduling*, 21(2):251–267, avril 2018.

[74] Andrew USELTON, Mark HOWISON, Nicholas J WRIGHT, David SKINNER, Noel KEEN, John SHALF, Karen L KARAVANIC et Leonid OLIKER : Parallel i/o performance: From events to ensembles. *In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[75] Erick D. WIKUM, Donna C. LLEWELLYN et George L. NEMHAUSER : One-machine generalized precedence constrained scheduling problems. *Operations Research Letters*, 16(2):87 – 99, 1994.

[76] Guangwei WU, Jianer CHEN et Jianxin WANG : On scheduling two-stage jobs on multiple two-stage flowshops. *arXiv preprint arXiv:1801.09089*, 2018.

[77] Bing XIE, Jeffrey CHASE, David DILLOW, Oleg DROKIN, Scott KLASKY, Sarp ORAL et Norbert PODHORSZKI : Characterizing output bottlenecks in a supercomputer. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 8. IEEE Computer Society Press, 2012.

[78] Li XU, Yueyao WANG, Thomas LUX, Tyler CHANG, Jon BERNARD, Bo LI, Yili HONG, Kirk CAMERON et Layne WATSON : Modeling i/o performance variability in high-performance computing systems using mixture distributions. *Journal of Parallel and Distributed Computing*, 139:87 – 98, 2020.

[79] Lianghong XU, James CIPAR, Elie KREVAT, Alexey TUMANOV, Nitin GUPTA, Michael A KOZUCH et Gregory R GANGER : Springfs: bridging agility and performance in elastic distributed storage. *In FAST*, pages 243–255, 2014.

[80] Orcun YILDIZ, Matthieu DORIER, Shadi IBRAHIM, Robert ROSS et Gabriel ANTONIU : On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. *In IPDPS 2016 - The 30th IEEE International Parallel and Distributed Processing Symposium*, pages 750–759, Chicago, United States, mai 2016.

[81] Xuechen ZHANG, Kei DAVIS et Song JIANG : Opportunistic data-driven execution of parallel programs for efficient i/o services. *In Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 330–341. IEEE, 2012.

[82] Zhou ZHOU, Xu YANG, Dongfang ZHAO, Paul RICH, Wei TANG, Jia WANG et Zhiling LAN : I/o-aware batch scheduling for petascale computing systems. *In Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 254–263. IEEE, 2015.

Publications

Articles in Peer-reviewed Conferences

[83] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E Singh, and Nicolas Vidal. Mapping and scheduling hpc applications for optimizing i/o. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

Articles in Peer-reviewed Journals

[84] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. Scheduling periodic i/o access with bi-colored chains: models and algorithms. *Journal of Scheduling*, pages 1–13, 2021.

Code

[85] Nicolas Vidal. Simulation code. https://gitlab.inria.fr/nividal/simulation-code.git, 2021.