HAL
open science

Multi-Purpose Efficient Resource Allocation for Parallel Systems

Fernando Mendonca

To cite this version:

Fernando Mendonca. Multi-Purpose Efficient Resource Allocation for Parallel Systems. Computer science. Université Grenoble Alpes, 2017. English. NNT : 2017GREAM021 . tel-01681424v2

HAL Id: tel-01681424

https://theses.hal.science/tel-01681424v2

Submitted on 12 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Fernando MENDONCA

Thèse dirigée par **Denis TRYSTRAM**, Professeur, Grenoble INP

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Politiques polyvalentes et efficientes d'allocation de ressources pour les systèmes parallèles

Multi-Purpose Efficient Resource Allocation for Parallel Systems

Thèse soutenue publiquement le **23 mai 2017**,
devant le jury composé de :

Monsieur PASCAL BOUVRY
PROFESSEUR, UNIVERSITE DU LUXEMBOURG, Rapporteur
Monsieur FREDERIC GUINAND
PROFESSEUR, UNIVERSITE LE HAVRE NORMANDIE, Président
Monsieur PHILIPPE OLIVIER A. NAVAUX
PROFESSEUR, UFRGS - BRESIL, Rapporteur
Monsieur DENIS TRYSTRAM
PROFESSEUR, GRENOBLE INP, Directeur de thèse
Monsieur GUILLAUME MERCIER
MAITRE DE CONFERENCES, INSTITUT POLYTECHNIQUE BORDEAUX, Examinateur
Monsieur FREDERIC SUTER
CHARGE DE RECHERCHE, CNRS DELEGATION RHONE AUVERGNE, Examinateur
Monsieur GIORGIO LUCARELLI
CHARGE DE RECHERCHE, INRIA DELEGATION ALPES, Examinateur

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions: Design Efficient Scheduling Policies 5
1.2.1 Fast Biological Sequence Comparison on Heterogeneous Platforms 5
1.2.2 A New Online Method for Scheduling Independent Tasks 6
1.2.3 Contiguity and Basic Locality in Backfilling Schedule . 7
1.2.4 Improving Backfilling with Full Locality Awareness . . 8
1.3 Organization of the text 9

2 Fast Biological Sequence Comparison on Heterogeneous Platforms **11**
2.1 Introduction 11
2.2 Biological Sequence Comparison 12
2.2.1 Presentation of the core problem 12
2.2.2 Smith-Waterman (SW) Algorithm 13
2.2.3 Parallelizing SW 14
2.3 Scheduling Algorithm with Dual Approximation 16
2.4 Designing the SWDUAL implementation 21
2.5 Experimental Results 22
2.5.1 Comparison to other implementations 23
2.5.2 Comparison to 5 genomic databases 24
2.5.3 Comparison of homogeneous and heterogeneous sets . 26
2.6 Conclusion 28

3 A New Online Method for Scheduling Independent Tasks **29**
3.1 Introduction 29
3.2 Definitions and notations. 31
3.3 Related works 32

3.4 Algorithms . . . 34
3.4.1 Standard scheduling policies . . . 34
3.4.2 Scheduling with redirections . . . 35
3.4.3 Scheduling with random redirections . . . 37
3.4.4 Dispatching policy . . . 38
3.5 Experimental results . . . 39
3.5.1 Construction of instances . . . 39
3.5.2 Experiments . . . 41
3.6 Conclusion . . . 49

4 Contiguity and Locality in Backfilling Scheduling 50
4.1 Introduction . . . 50
4.2 Related Works . . . 52
4.3 Problem Setting . . . 52
4.4 Worst-case Guarantees for Optimal Solutions . . . 54
4.4.1 Contiguous vs. Non-contiguous . . . 54
4.4.2 Local vs. Non-local . . . 55
4.4.3 Topological-aware vs. Basic Model . . . 57
4.5 Allocation Algorithms . . . 60
4.5.1 Backfilling . . . 60
4.5.2 Algorithms . . . 64
4.6 Experimental Results . . . 67
4.6.1 Makespan . . . 69
4.6.2 Contiguity and Locality . . . 71
4.7 Conclusion . . . 73

5 Improving Backfilling with Full Locality Awareness 76
5.1 Introduction . . . 76
5.2 Related Work . . . 78
5.3 Problem Modeling . . . 79
5.3.1 Platform . . . 80
5.3.2 Backfilling . . . 81
5.3.3 Success Rate . . . 81
5.3.4 Overhead Model . . . 81
5.4 Online Scheduling . . . 82
5.5 Job reassignment . . . 83
5.6 Allocation Algorithms . . . 85
5.6.1 Basic Backfilling . . . 85
5.6.2 Basic Locality . . . 86
5.6.3 Contiguity . . . 87

5.6.4 Platform Level (Algorithm 5) 88
5.7 Experimental Results 90
5.7.1 Goals 90
5.7.2 Simulator 90
5.7.3 Parameters 92
5.7.4 Platforms 93
5.7.5 Experimental Analysis 93
5.8 Conclusion 106

6 Conclusion **107**

Abstract

The field of parallel supercomputing has been changing rapidly in recent years. The reduction of costs of the parts necessary to build machines with multicore CPUs and accelerators such as GPUs are of particular interest to us. This scenario allowed for the expansion of large parallel systems, with machines far apart from each other, sometimes even located on different continents. Thus, the crucial problem is how to use these resources efficiently.

In this work, we first consider the efficient allocation of tasks suitable for CPUs and GPUs in heterogeneous platforms. To that end, we implement a tool called SWDUAL, which executes the Smith-Waterman algorithm simultaneously on CPUs and GPUs, choosing which tasks are more suited to one or another. Experiments show that SWDUAL gives better results when compared to similar approaches available in the literature.

Second, we study a new online method for scheduling independent tasks of different sizes on processors. We propose a new technique that optimizes the stretch metric by detecting when a reasonable amount of small jobs is waiting while a big job executes. Then, the big job is redirected to separate set of machines, dedicated to running big jobs that have been redirected. We present experiment results that show that our method outperforms the standard policy and in many cases approaches the performance of the preemptive policy, which can be considered as a lower bound.

Next, we present our study on constraints applied to the Backfilling algorithm in combination with the FCFS policy: Contiguity, which is a constraint that tries to keep jobs close together and reduce fragmentation during the schedule, and Basic Locality, that aims to keep jobs as much as possible inside groups of processors called clusters. Experiment results show that the benefits of using these constrains outweigh the possible decrease in the number of backfilled jobs due to reduced fragmentation.

Finally, we present an additional constraint to the Backfilling algorithm called Full Locality, where the scheduler models the topology of the platform as a fat tree and uses this model to assign jobs to regions of the platform where communication costs between processors is reduced. The experiment campaign is executed and results show that Full Locality is superior to all the previously proposed constraints, and specially Basic Backfilling.

Résumé

Les plateformes de calcul à grande échelle ont beaucoup évolué ces dernières années. coûts des composants simplifie la construction de machines possédant des multicœurs et des accélérateurs comme les GPUs. Ceci a permis une propagation des plateformes à grande échelle, dans lesquelles les machines peuvent être éloignées les unes des autres, pouvant même être situées sur différents continents. Le problème essentiel devient alors d'utiliser ces ressources efficacement.

Dans ce travail nous nous intéressons d'abord à l'allocation efficace de tâches sur plateformes hétérogènes composées CPU et de GPUs. Pour ce faire, nous proposons un outil nommé SWDUAL qui implémente l'algorithme de Smith-Waterman simultanément sur CPU et GPUs, en choisissant quelles tâches il est plus intéressant de placer sur chaque type de ressource. Nos expériences montrent que SWDUAL donne de meilleurs résultats que les approches similaires de l'état de l'art.

Nous analysons ensuite une nouvelle méthode d'ordonnancement en ligne de tâches indépendantes de différentes tailles. Nous proposons une nouvelle technique qui optimise la métrique du *stretch*. Elle consiste à déplacer les jobs qui retardent trop de petites tâches sur des machines dédiées. Nos résultats expérimentaux montrent que notre méthode obtient de meilleurs résultats que la politique standard et qu'elle s'approche dans de nombreux cas des résultats d'une politique préemptive, qui peut être considérée comme une borne inférieure.

Nous nous intéressons ensuite à l'impact de différentes contraintes sur la politique FCFS avec backfilling. La contrainte de contiguïté essaye de compacter les jobs et de réduire la fragmentation dans l'ordonnancement. La contrainte de localité basique place les jobs de telle sorte qu'ils utilisent le plus petit nombre de groupes de processeurs appelés *clusters*. Nos résultats montrent que les bénéfices de telles contraintes sont suffisants pour compenser la réduction du nombre de jobs backfillés due à la réduction de la fragmentation.

Nous proposons enfin une nouvelle contrainte nommée localité totale, dans laquelle l'ordonnanceur modélise la plateforme par un *fat tree* et se sert de cette information pour placer les jobs là où leur coût de communication est minimal. Notre campagne d'expériences montre que cette contrainte obtient de très bons résultats par rapport à un backfilling basique, et de meilleurs résultats que les contraintes précédentes.

Chapter 1

Introduction

The field of parallel supercomputing has been changing rapidly in recent years, particularly after the introduction of multicore and specialized architectures. Such processing units, for example Graphical Processing Units (GPUs), Field-programmable gate arrays (FPGAs) and Intel Xeon Phis, feature hardware that is much more suited for computing regular parallel jobs, when comparing to the previous standards. Indeed, the amount of cores available within each system is much higher and this additional computing power comes with its own challenges. Furthermore, this scenario has allowed for the creation and expansion of large parallel systems, with machines far apart from each other, sometimes even located on different continents. Because the distance between these clusters is high, there is also high latency, even though these machines are usually connected by high performance networks. Thus, the communication performance can be dramatically affected by the location of the processes within the system since the closer the processes are from each other, the fastest the communication will be due to latency. Beyond that, we can safely say that the size of parallel platforms is increasing rapidly as machines get cheaper to build.

1.1 Motivation

One good example of a platform where a large number of nodes are available but geographically distant from each other is the Grid 5000 testbed [2]. It is supported by a scientific interest group involving Inria, CNRS, RENATER and several Universities as well as other organizations. The grid provides access to nodes in eight different locations in France, as shown in Figure 1.1, consisting of 1000 nodes, 8000 cores, grouped in homogeneous

clusters. The nodes themselves are heterogeneous, containing not only cores but also coprocessors like Graphic Processing Units (GPUs) and Xeon Phi boards.

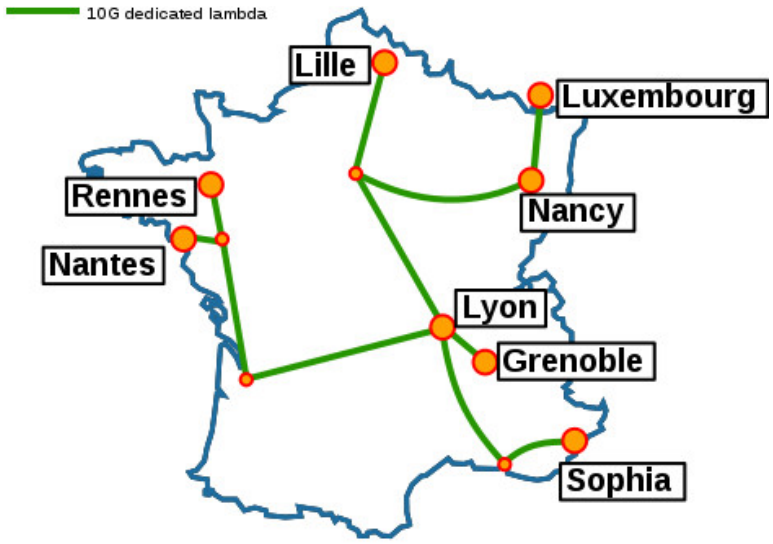


Figure 1.1 – Grid 5000 high level network topology, taken from the official website on 21 March 2017

One of the problems with such platforms is that although network conditions between clusters of the same site tend to be favorable, the same cannot be said between sites, specially if they are geographically far from each other. In such cases, it is common that communication between sites have high latency, even though the links may be dedicated and thus have high bandwidth.

Even if we consider only machines located in clusters from the same site, a case can be made for assigning jobs to processors that are as close together as possible. The reason is that the topology is most likely shaped like a tree, where machines are placed in racks, which in turn are connected to each other through network nodes on different levels. In this case, we can agree that running a job on two processors in the same machine is considerably faster than running the same job on two processors located in two different clusters, with several levels of network between them.

Figure 1.2 shows an example of a schedule through a Gantt chart. The horizontal axis represents time and the vertical axis the processors available in the platform. In such platforms, a large number of users can submit

jobs, which are executed in partitions of the platform, that is, subsets of processors.

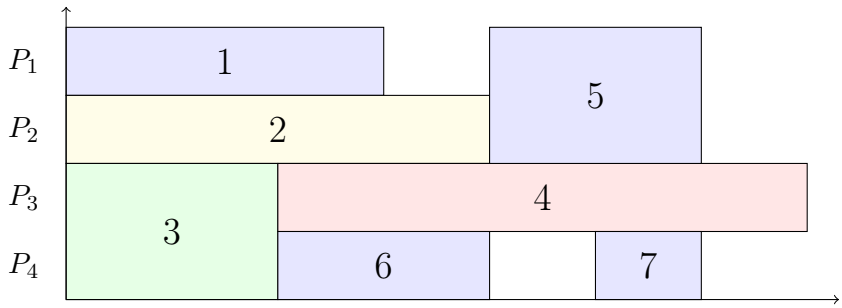


Figure 1.2 – Schedule representation using a Gantt chart

The most basic scheduling scheme that can be used in distributed-memory parallel supercomputers is variable partitioning, where each job receives a partition of the machine with its desired number of processors. Such partitions are allocated in a First Come, First Served (FCFS) fashion. The problem with this approach is that it suffers from fragmentation, where processors are available but cannot be used until much later in the schedule. As a result system utilization can be penalized.

Figure 1.3 shows an example of schedule done with the FCFS scheme. In this case, job number 6 is submitted and can not be moved forward.

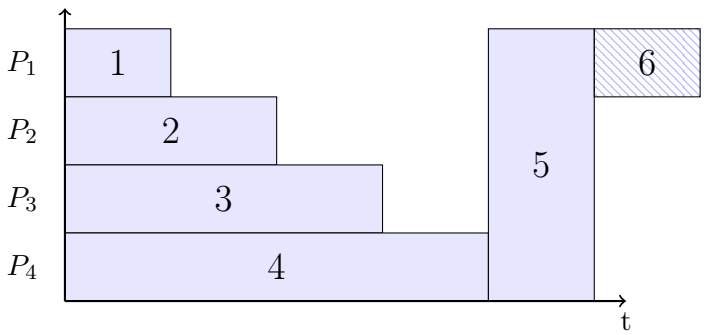


Figure 1.3 – Example of FCFS allocation scheme

A better solution to this problem is to require users to estimate the run time of their jobs. Using this information, small jobs can be moved forward in the schedule and fill gaps left over from other jobs. This approach is called Backfilling and was developed for the IBM SP1 parallel supercomputer installed at the Argonne National Laboratory [51]. Users are expected to

submit jobs in conjunction with accurate estimates of their job's run time, since underestimating the job's run time may lead to the job being killed before it finishes executing, while overestimating may lead to long wait times and likely low utilization of processors.

Figure 1.4 shows an example of a schedule done with Backfilling. We can see that job 5, being the last one to be submitted, was allowed to move forward and start executing right after job 1.

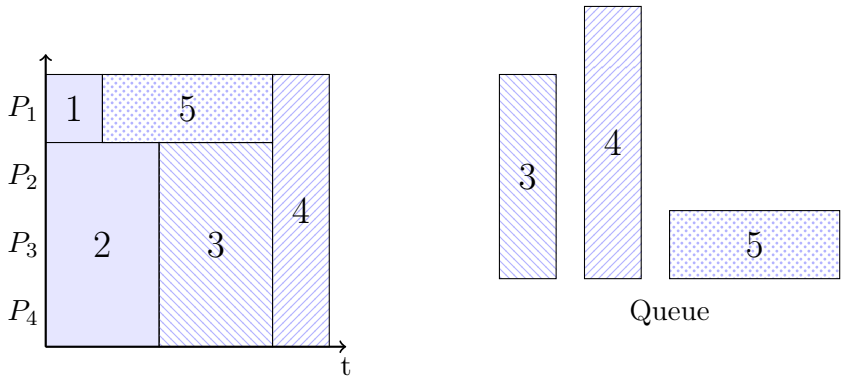


Figure 1.4 – Example of allocation using Conservative backfilling

The Backfilling algorithm, combined with a First Come, First Served (FCFS) policy, works very well when there are no latency issues between the processors. On the other hand, Backfilling shows significantly worst results if the latency between some of the processors available is high enough to impact the jobs' run times. The reason for this poor performance is that the Backfilling algorithm does not choose specific processors for each job, but rather just picks the first ones it encounters that are available for the duration of the job.

As a result, scheduling jobs with Backfilling can be largely improved by inserting additional constraints that help better choose which processors each job should be assigned to.

Moreover, one other problem with scheduling jobs utilizing the FCFS scheme is that it may happen that a big job is submitted and starts executing right before a large number of small jobs are submitted. As a consequence, most or all the small jobs wait have to wait for a long time in the system, until the big job finishes. This phenomenon happens even if a scheme different than FCFS is being used. For example, this can happen if jobs are sorted by increasing run time when they are submitted. If only the big job is submitted, it is allowed to start executing. Then, if a large number of small

jobs is submitted, the same situation happens.

Figure 1.5 shows an example of a schedule where a job with a very long run time is executing while several jobs with short run times are waiting.

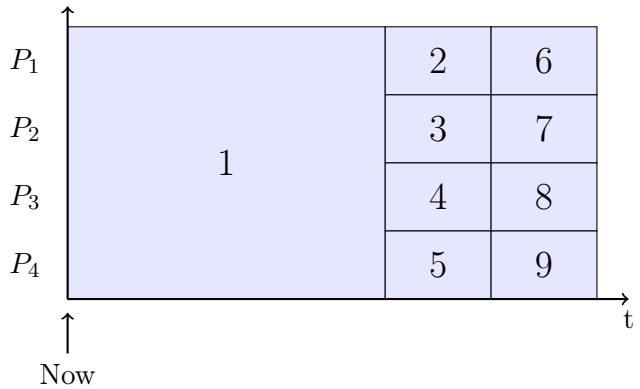


Figure 1.5 – Example of a schedule where a big job is executing while several small jobs have to wait

Furthermore, it is common that platforms provide nodes with more than one type of processing unit, including processors, GPUs, Field-programmable gate arrays (FPGAs) and Xeon Phi accelerators. In such cases, it is essential that the platform's characteristics are utilized as efficiently as possible. The way this can be done is by matching the job's tasks to the different types of processing units available, depending on processing power, availability, architecture, among others. Also, it is important to consider that some tasks may be more suited to specific types of processing units but not others, or may even be executed exclusively on a particular type of processing unit.

1.2 Contributions: Design Efficient Scheduling Policies

1.2.1 Fast Biological Sequence Comparison on Heterogeneous Platforms

There are some applications that require considerable amount of time to complete with limited resources. As parallel platforms get more accessible, it becomes more interesting to exploit these platforms and utilize resources in a parallel fashion. Additionally, heterogeneous platforms propose a new challenge for running parallel applications efficiently. It is not just a matter

of running jobs whenever a processing unit is available, but also consider their differences in architecture and performance.

Since heterogeneous platforms are usually composed of accelerators connected to multi-core hosts, we study how to efficiently utilize both the accelerators and the processors to execute the Smith-Waterson (SW) algorithm [62] in parallel. There are some approaches in the literature that explore such idea, but they assume that multi-core processors and accelerators have similar processing powers, distribute work proportionally in a static fashion considering the theoretical processing power of each type of processing unit or assign one task at a time in a Self-Scheduling strategy.

To achieve this goal, we implement a tool called SWDUAL, which executes the Smith-Waterson algorithm on hybrid platforms composed of multi-core processors and GPUs. Furthermore, SWDUAL is based on a fast dual approximation scheduling algorithm that selects the most suitable tasks to be executed on the GPUs while balancing the load over the whole platform.

Experiments are prepared and executed to demonstrate that SWDUAL gives better results when comparing to other approaches available in the literature. We show that our approach was able to significantly reduce the execution time of the sequence database searches using the Smith-Waterman algorithm, as well as finish the execution with almost no idle time on most of the included processors and GPUs.

This work was published in the International Conference on Parallel Processing (ICPP) in 2014 [37], under the name "Fast Biological Sequence Comparison on Hybrid Platforms".

1.2.2 A New Online Method for Scheduling Independent Tasks

When sequential jobs of different run times are being executed in a platform composed of a set of processors, a phenomenon can happen where big jobs start first and small jobs that are submitted later are required to wait for a long time. This scenario is particularly bad for metrics like *flow* or *stretch*, that measure the amount of time each job stays in the system, since a large number of small jobs wait while a few big jobs are being executed.

We propose a novel technique for scheduling sequential jobs in this fashion that optimizes the stretch metric. The main idea is to detect when a reasonable number of small jobs is waiting for a big job and redirect the big job to a separate set of processors dedicated to running redirected jobs.

This technique is implemented in a discrete event simulator. Jobs from real traces are converted to sequential independent jobs and grouped to-

gether in periods of one week each. Then, different numbers of periods are chosen randomly depending on the characteristics of the platforms included in the experiments. We use these generated traces to execute the experiments and show that our method outperforms the standard policy and in many cases approaches the performance of the preemptive policy, which can be considered as a lower bound.

This work was published in the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) in 2017 [45], under the name "A new on-line method for scheduling independent tasks".

1.2.3 Contiguity and Basic Locality in Backfilling Schedule

Backfilling is a scheduling algorithm that can move small jobs forward in the schedule in order to fill gaps. Also, it shows good results by itself and is quite hard to improve, although not impossible.

Attempting to improve the Backfilling algorithm can introduce delays to the jobs and reduce the effectiveness of the improvements being made. On the other hand, one of the weak points of Backfilling is that it does not choose any particular processors when assigning a job. It picks the first ones available, which can have a different effect depending on the implementation. For this reason, it is specially interesting to explore additional constraints made to this phase of the algorithm, helping it better select processors for each job.

Contiguity is a basic constraint that attempts to keep jobs close together and reduce fragmentation during a schedule. This is done by assigning jobs to contiguous blocks of processors, which are numbered. On one hand, employing this constraint might reduce the chance of the algorithm actually moving small jobs forward since there will be less space available. On the other hand, the reduced fragmentation tends to facilitate the assignment of larger jobs in the first place, helping improve different metrics.

Figure 1.6 shows an example of a platform as modeled by the Contiguity constraint.

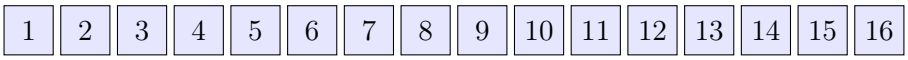


Figure 1.6 – Example a of a platform as utilized by the Contiguity constraint

Basic Locality is a constraint that aims to keep jobs close together but with a different organization. It divides the available processors in groups called clusters. The size of each cluster is configurable. The goal is then to

assign jobs in such a way that as few clusters as possible are occupied by the job. This in turn helps in similar ways as contiguity, reducing fragmentation, although not as much. Additionally, in a scenario where communication costs are not negligible, associating the clusters with the organization of the network helps improve the schedule further.

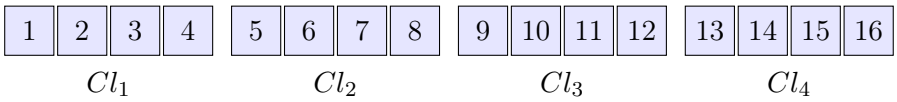


Figure 1.7 – Example a of a platform as utilized by the Basic Locality constraint

Figure 1.7 shows an example of a platform modeled by the basic locality constraint. Each cluster, from Cl_1 to Cl_4, consists of four processors.

These two constraints are used to improve the efficiency of the Backfilling algorithm. To demonstrate this, a new discrete event simulator is developed. Experiments are then designed focusing on employing real traces and utilizing the jobs' real run time, since user submitted times are strongly overestimated. For this reason, the scheduler runs in an offline fashion and we focus on the completion time metric (C_{max}). Then, subsets of jobs are taken from random starting points in the original traces and used as input in the experiments.

Results show that the benefits of these constraints indeed outweigh the possible decrease in the number of jobs backfilled due the reduced fragmentation. Particularly, we demonstrate that contiguity is able to achieve some level of locality without knowledge of the platform. We also note that these algorithms can be easily integrated into existing batch schedulers as plugins.

This work was published in the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) 2015 [46], under the name "Contiguity and Locality in Backfilling Scheduling".

1.2.4 Improving Backfilling with Full Locality Awareness

Full Locality is a constraint that considers the topology of the platform, and not just the size of each group of processors at the last level. The main goal is to use this information to assign jobs to regions of the platform where communication costs due to latency are minimized. This constraint does not require jobs to be assigned contiguously, since communication costs inside the same group of processors is the same for all of them. As a consequence, there is but a slight reduction in fragmentation and consequently in the

number of backfilled jobs. For this reason, full locality is superior to all the other constraints, because it focuses on reducing the run time by minimizing communication costs while keeping the Backfilling algorithm's efficiency as intact as possible.

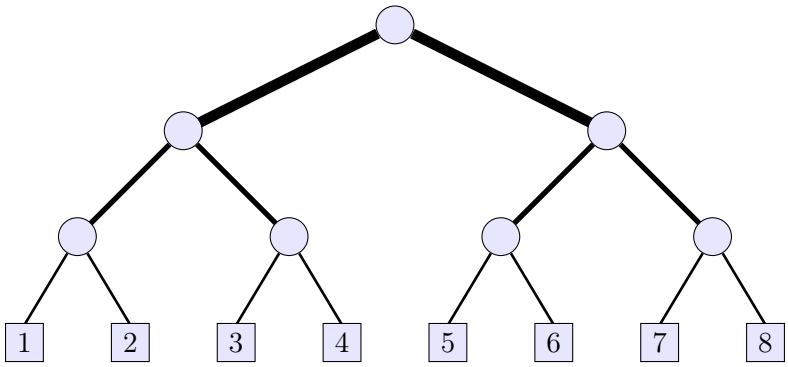


Figure 1.8 – Example of a platform as utilized by the full locality constraint

In order to compare the Full Locality constraint to the previously proposed ones, it was added to the simulator. Then, new experiments were designed, now focusing on utilizing the original traces with as few modifications as possible. In this instance, the only modification done to the traces is the division in periods of one week. Then, each instance is composed by one or more contiguous periods chosen at random. This is done to show that the results represent different scenarios relevant to each of the included platforms.

Thus, we keep the jobs' original information and consider both user submitted and real run times. For this reason, the scheduler works in an online fashion, where it has no information about the subsequent jobs and does not know when the currently running jobs will finish. This information is restricted to the part of the code that deals with the submission and ending events. The experiments are executed and generated results show that Full Locality is indeed superior to all the previous variants, for several metrics.

At time of writing this work has not been submitted to any conference. However, we have plans to submit it to to a conference in the following months.

1.3 Organization of the text

The Chapters of the text are organized in the following way:

Chapter 2 presents the sequence comparison problem, the strategy and its implementation and experimental results.

Chapter 3 contains the definition of the studied problem, a presentation of the base-line non-preemptive algorithms used as well as the preemptive lower bounds used in the experiments. Additionally, the algorithms based on heavy tasks and redirections are proposed. Finally, this Chapter describes the organization of experimental phase and provide results.

Chapter 4 discusses the most relevant related works concerning the scheduling problem, presents the theoretical analysis of the degradation ratio due to additional topology constraints applied to the Backfilling algorithm and proposed allocation algorithms targeting locality. Finally, the chapter reports the results of the conducted experiments.

Lastly, Chapter 5 introduces an update to the discrete event simulator presented in the previous Chapter. This update contains additional features such as job reassignment. Furthermore, this new version of the simulator includes an improved model of the platform topology as well as the implementation of a new type of constraint that is aware of the topology. Next, the Chapter discusses the configuration of the conducted experiments and presents the results.

Chapter 2

Fast Biological Sequence Comparison on Heterogeneous Platforms

2.1 Introduction

The goal of this Chapter is to design an efficient implementation of the classical problem of comparing biological sequences in a parallel multi-core platform with hardware accelerators.

Once a new biological sequence is discovered, its functional/structural characteristics must be established. In order to do that, the newly discovered sequence is compared against other sequences, looking for similarities. Sequence comparison is, therefore, one of the most crucial operations in Bioinformatics [50]. The most accurate algorithm to execute pairwise comparisons is the one proposed by Smith and Waterman (SW) [62], which is based on dynamic programming and runs in quadratic time and space complexity to the length of the sequences. This can easily lead to very large execution times and huge memory requirements, since the size of biological databases is growing exponentially.

Parallel implementations can be used to compute results faster, reducing significantly the time needed to obtain results with the SW algorithm. Indeed, many proposals exist to execute SW on clusters [55, 16] and computational grids [18]. More recently, hardware accelerators such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays) have been explored to speed-up the SW algorithm [20, 44, 34]. In addition to that, SIMD extensions of general-purpose processors, such as the Intel SSE,

have also been explored to accelerate SW implementations [57].

Since accelerators are usually connected to a multi-core host, the idea is to use both the accelerators and the CPUs to execute SW in parallel. There are some approaches in the literature that explore such idea [59, 60, 49]. In order to distribute work among the hybrid processing elements, these approaches usually assume that multi-cores and accelerators have the same processing power [60], distribute work proportionally, considering the theoretical computing power of each processing element [49] or assign one work unit at the time [59] in a Self-Scheduling strategy.

In this Chapter, we propose SWDUAL, a new implementation of the Smith-Waterman algorithm for hybrid platforms composed of multiple processors and multiple GPUs. SWDUAL is based on a fast dual approximation scheduling algorithm that selects the most suitable tasks to be run on the GPUs while keeping a good balance of the computational load over the whole platform [38].

Given a set of query sequences and a biological database, our strategy uses a one round master-slave approach to assign tasks to the processing elements according to the dual approximation scheduling algorithm being used.

The remainder of this Chapter is organized as follows. Section 2.2 presents the sequence comparison problem and recalls the principle of the classical SW algorithm. The strategy and its implementation for executing SW on hybrid platforms are proposed respectively in Sections 2.3 and 2.4. Section 2.5 presents the experimental results. Finally, Section 2.6 presents a conclusion for the Chapter.

2.2 Biological Sequence Comparison

2.2.1 Presentation of the core problem

A biological sequence is a structure composed of nucleic acids or proteins. It is represented by an ordered list of residues, which are nucleotide bases (for DNA or RNA sequences) or amino acids (for protein sequences).

DNA and RNA sequences are treated as strings composed of elements of the alphabets $\sum = \{A, T, G, C\}$ and $\sum = \{A, U, G, C\}$, respectively. Protein sequences are also treated as strings which elements belong to an alphabet with, normally, 20 amino acids.

Since two biological sequences are rarely identical, the sequence comparison problem corresponds to approximate pattern matching. To compare two sequences, a good alignment between each other should be determined.

This corresponds to place one sequence above the other making clear the correspondence between similar characters [50]. In an alignment, some gaps (space characters) can be inserted in arbitrary locations such that the sequences end up with the same size.

Given an alignment between sequences s and t, a score is associated to it as follows. For each two bases in the same column:

- a punctuation ma is associated if both characters are identical ($match$);
- a penalty mi, if the characters are different ($mismatch$);
- a penalty g, if one of the characters is a gap.

The score is obtained by the addition of all these values. The maximal score is called the similarity between the sequences. Figure 2.1 presents one possible global alignment between two DNA sequences and its associated score. In this example, $ma = +1$, $mi = -1$ and $g = -2$.

$$\underbrace{\begin{array}{ccccccccc} A & C & T & T & G & T & C & C & G \\ A & - & T & T & G & T & C & A & G \\ \hline +1 & -2 & +1 & +1 & +1 & +1 & +1 & -1 & +1 \end{array}}_{score = 4}$$

Figure 2.1 – Example of an alignment and score

2.2.2 Smith-Waterman (SW) Algorithm

The SW algorithm [62] is an exact method based on dynamic programming to obtain the optimal pairwise local alignment in quadratic time and space in the length of the sequences.

The first phase of the SW algorithm starts by two input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ is the size of sequence s. The similarity matrix is denoted by $H_{m+1,n+1}$, where $H_{i,j}$ contains the score between prefixes $s[1..i]$ and $t[1..j]$. At the beginning, the first row and column are filled with zeros. The remaining elements of H are obtained from Equation (2.1). In addition, each cell $H_{i,j}$ contains the information about the cell that was used to produce the value. $S_{i,j}$ is a similarity score for the elements i and j

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i,j-1} + g \\ H_{i-1,j} + g \\ 0 \end{cases} \tag{2.1}$$

The SW algorithm assigns a constant cost to gaps. Nevertheless, in nature, gaps tend to appear in groups. For this reason, a higher penalty is usually associated to the first gap and a lower penalty is given to the following ones (this is known as the affine-gap model). Gotoh [28] proposed an algorithm based on SW that implements the affine-gap model by calculating three Dynamic Programming (DP) matrices, namely H, E and F, where E and F keep track of gaps in each of the sequences. The gap penalties for starting and extending a gap are G_s and G_e, respectively. This recursion formulas are given by Equations (2.2), (2.3) and (2.4).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \tag{2.2}$$

$$E_{i,j} = -G_e + \max \begin{cases} E_{i,j-1} \\ H_{i,j-1} - G_s \end{cases} \tag{2.3}$$

$$F_{i,j} = -G_e + \max \begin{cases} F_{i-1,j} \\ H_{i-1,j} - G_s \end{cases} \tag{2.4}$$

2.2.3 Parallelizing SW

There are several ways to parallelize the SW algorithm. The following paragraph describes the comparison of a set q of m query sequences $(q_1, q_2, ..., q_m)$ to a set d of n database sequences $(d_1, d_2, ..., d_n)$. It is assumed that the size of the database is much larger than the set of query sequences ($m \ll n$).

In the fine-grained approach, the comparison of one query sequence and one database sequence (i.e. a single SW execution) is done by several Processing Elements (PEs). The data dependency in the matrix calculation is non-uniform, and the calculations that can be done in parallel evolve as waves on diagonals (according to Equation (2.2)). Figure 2.2 illustrates a

fine-grained column-based block partition technique with four PEs. At the beginning, only p_0 is computing. When p_0 finishes calculating the values of a block of matrix cells, it sends its border column to p_1, that can start calculating and so on. Note that this solution may be unbalanced: very close to the end of the matrix computation, only p_3 is calculating. When the PEs finish to compare q_1 to d_1, they start comparing q_1 to d_2 and so on, until the comparison of q_m to d_n is completed.

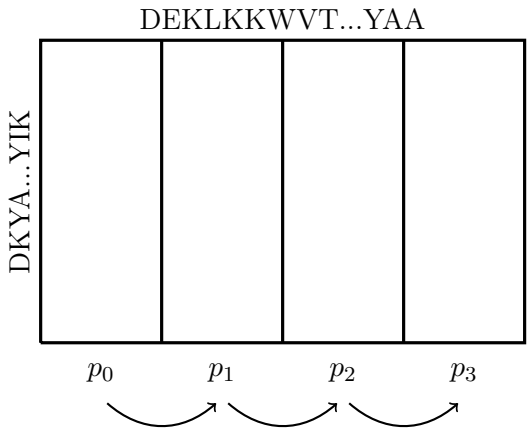


Figure 2.2 – Fine-grained strategy to parallelize the SW algorithm

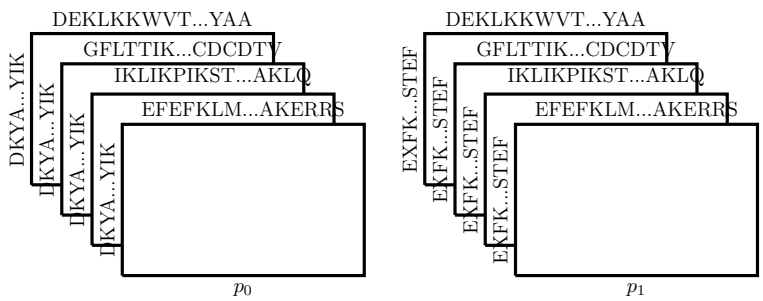


Figure 2.3 – Very coarse-grained strategy to parallelize the SW algorithm

In the very coarse-grained approach, each PE compares a different query sequence to the whole database (see Figure 2.3). For instance, p_0 compares q_1 to d, p_1 compares q_2 to d and so on. Note that, in this case, the number of SW comparisons executed by each processing element is big and this approach can easily lead to load imbalance.

The SWDUAL implementation uses both the fine-grained and very coarse-

grained approach. Each one of the workers uses fine-grained approach to accelerate the execution of the SW algorithm for a particular comparison. That approach is dependent on the type of worker and the techniques being used to optimize each comparison. At the same time, other workers are comparing other sequences of the query set to the database in the same way. In our case the master uses the scheduling algorithm to allocate tasks to the workers. Each task is equivalent to the comparison of one task of the query set to the whole database.

In the following section, we describe the scheduling algorithm used by the master to allocate the tasks to the workers.

2.3 Scheduling Algorithm with Dual Approximation

In the implementation targeted in this Chapter, tasks are pairwise comparisons of two sequences. The problem is to determine an allocation of the tasks to the GPUs that minimizes the global completion time [1].

The principle of the proposed scheduling algorithm is to use the dual approximation technique introduced in [30] and which is recalled as follows. A g-dual approximation algorithm for any minimization problem takes a real number λ (called the guess) as an input and either delivers a schedule whose makespan is at most $g\lambda$ or answers correctly that there exists no schedule of length at most λ.

We target $g = 2$. Let λ be the current real number input for the dual approximation. In the following, we assert that there exists a schedule of length lower than λ. Then, we have to show how it is possible to build a schedule of length at most 2λ.

We introduce an allocation function $\pi(j)$ of a task T_j which corresponds to the processor where the task is processed. The set $\mathcal{C}$ (resp. $\mathcal{G}$) is the set of all the CPUs (resp. GPUs). Therefore, if a task T_j is assigned to a CPU, we can write $\pi(j) \in \mathcal{C}$. Each task T_j has two processing times, $\overline{p_j}$ if it is processed on a CPU, $\underline{p_j}$ if it is processed on a GPU. We define W_C as being the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the processing times of the tasks allocated to the CPUs: $W_C = \sum_{j \,/\, \pi(j) \in \mathcal{C}} \overline{p_j}$.

To take advantage of the dual approximation paradigm, we have to make explicit the consequences of the assumption that there exists a schedule of

[1] also called makespan, or C_{max}

length at most λ. We state below some basic properties of such a schedule:

- The execution time of each task is at most λ.
- The computational area on the CPUs is at most $m\lambda$.
- The computational area on the GPUs is at most $k\lambda$.

We are looking for an assignment of the tasks to either a CPU or a GPU satisfying the following two constraints:

- (C1) The total computational area W_C on the CPUs is at most $m\lambda$.
- (C2) The total computational area on the GPUs is lower than $k\lambda$.

We define for each task T_j a binary variable x_j such that $x_j = 1$ if T_j is assigned to a CPU or 0 if T_j is assigned to a GPU. Determining if an assignment satisfying (C_1) and (C_2) exists corresponds to solving a minimization knapsack problem [48] that can be formulated as follows:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{2.5}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} \underline{p_j} \left(1 - x_j\right) \leqslant k\lambda \tag{2.6}$$

$$x_j \in \{0, 1\} \quad \forall j = 1, \ldots, n \tag{2.7}$$

Equation (2.5) represents the minimal workload on all the CPUs. Constraint (2.6) imposes an upper bound on the computational area of the GPUs which is $k\lambda$ (cf. (C2)).

The knapsack is solved by a greedy algorithm. Usually the knapsack is a maximization problem. Here we consider the opposite minimization version. The tasks are sorted by decreasing order of the ratio $\frac{\overline{p_j}}{\underline{p_j}}$. Thus, the most prioritary tasks are those with the best relative processing times on GPUs. Figure 2.4 depicts the principle of the greedy knapsack. The knapsack allocates the first tasks to the GPUs until the computational area on the GPUs is roughly equal to $k\lambda$.

The result of the knapsack leads to a solution with a computational area on GPUs larger than $k\lambda$. All remaining tasks are scheduled on CPUs. If the value of the computational area on the CPUs is greater than $m\lambda$, then there

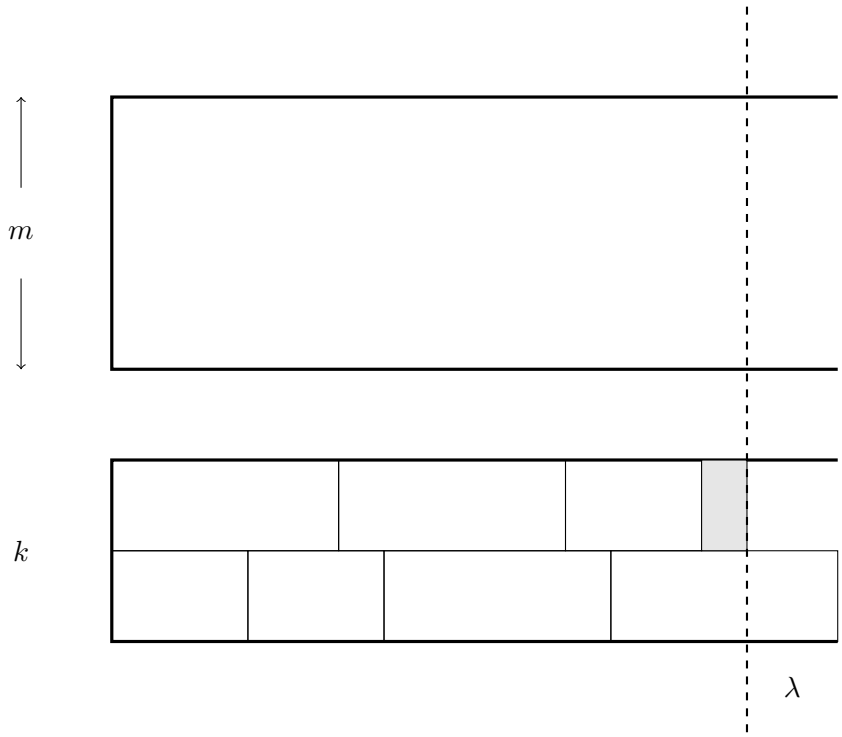


Figure 2.4 – Greedy knapsack fills the GPUs with tasks up to getting a computational area larger than $k\lambda$ on the GPUs

exists no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation.

The scheduling on the CPUs after the allocation of the greedy knapsack is done with a list scheduling algorithm assigning the tasks on an available processor of the corresponding type in the assignment (cf. Figure 2.5).

Proposition 1. *If W_C is lower than $m\lambda$, there exists a feasible solution with a makespan at most 2λ.*

Proof. The makespan on the CPUs, C_{max}^{CPU}, is bounded by the following inequality:

$$C_{max}^{CPU} \leqslant \max_{1 \leqslant j \leqslant n} (\overline{p_j} x_j) + \frac{\sum_{j=1}^{n} \overline{p_j} x_j}{\sum_{j=1}^{n} x_j} \tag{2.8}$$

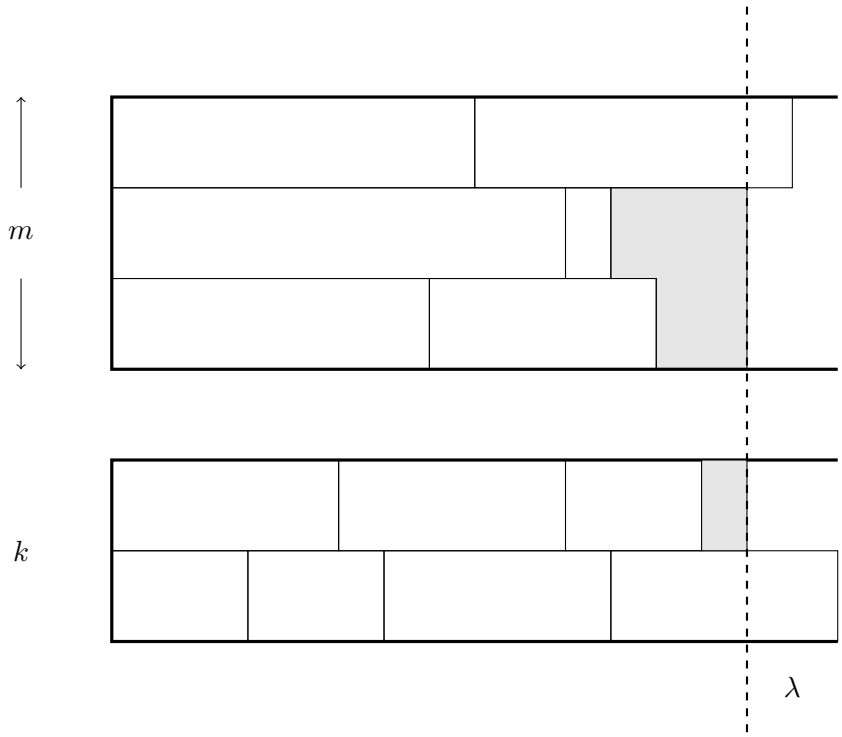


Figure 2.5 – List scheduling fills the CPUs with remaining tasks. The computational area is smaller than $m\lambda$, otherwise λ is smaller than C^*_{max}

All the tasks assigned to the CPUs have a processing time lower than λ, therefore $\max\limits_{1 \leqslant j \leqslant n} \overline{p_j} x_j \leqslant \lambda$ and $\sum\limits_{j=1}^{n} \overline{p_j} x_j \leqslant m\lambda$ with the hypothesis that W_C is lower than $m\lambda$. We obtain:

$$C^{CPU}_{max} \leqslant \left(1 + \frac{m}{\sum\limits_{j=1}^{n} x_j} \right) \lambda \tag{2.9}$$

Moreover, we can assume $\sum\limits_{j=1}^{n} x_j > m$, otherwise the optimal solution is straightforward (one task per CPU), thus:

$$C^{CPU}_{max} \leqslant 2\lambda \tag{2.10}$$

Let us turn now to the GPU side. Let $jlast$ be the index of the last task selected by the knapsack. The task $jlast$ is thus the last task scheduled by the greedy knapsack on the GPUs. Hence, task $jlast$ has no influence at all on the scheduling of all the other tasks.

Two cases hold (cf. Equation (2.11)): either the $jlast$ task is not the last to be completed or it is. On the first hand, $jlast$ can be removed from the schedule instance without changing the makespan. The computational area of all tasks except $jlast$ is smaller than $k\lambda$ thus the guarantee is the same as the one derived for the CPU schedule. On the second hand, the computational area of all tasks save $jlast$ is also smaller than $k\lambda$ thus, when the list algorithm schedules the $jlast$ task, the least loaded of the k GPUs is loaded less than λ. Hence the $jlast$ task ends before 2λ.

$$C_{max}^{GPU} \leqslant \begin{cases} \max\limits_{1 \leq j \leq n | j \neq jlast} \left(\underline{p_j}(1-x_j)\right) + \frac{\sum_{j=1}^{n} \underline{p_j}(1-x_j) - \underline{p_{jlast}}}{k} \leqslant 2\lambda \\ \left(\underline{p_j}(1-x_j)\right) + \frac{\sum_{j=1}^{n} \underline{p_j}(1-x_j) - \underline{p_{jlast}}}{k} \leqslant 2\lambda \end{cases} \tag{2.11}$$

Since the makespan of the schedule is the maximum of the makespans on the CPUs and on the GPUs, we get:

$$C_{max} \leqslant 2\lambda \tag{2.12}$$

□

We have described one step of the dual-approximation algorithm, with a fixed guess. A binary search will be used to try different guesses to approach the optimal makespan as follows.

Binary Search We first take an initial lower bound B_{min} and an initial upper bound B_{max} of our optimal makespan. We start by solving the problem with λ equal to the average of these two bounds and then we adjust the bounds:

- If the previous algorithm returns "NO", then λ becomes the new lower bound.
- If the algorithm returns a schedule of makespan at most 2λ, then λ becomes the new upper bound.

The number of iterations of this binary search can be bounded by $\log(B_{max} - B_{min})$.

Cost Analysis The greedy algorithm used to fill the GPUs only requires a sorting of the tasks, whereas the list scheduling used for the CPUs is linear. Therefore, the time complexity of each step of the binary search is $\mathcal{O}(n \log(n))$.

The algorithm described above returns a schedule with a makespan equal to at most twice the optimal makespan. Some constraints on the number of tasks with processing times larger than $\frac{2}{3}\lambda$, λ being the current guess, in the algorithm can be added to the original problem. The resolution of the knapsack problem with these additional constraints via dynamic programming can reduce the makespan of the schedule returned by the algorithm to $\frac{3}{2}OPT$, where OPT is the optimal makespan. This method is described in [38], and has a time complexity in $\mathcal{O}\left(n^2mk^2\right)$ per step of the binary search. This time complexity is important, but it can be lowered with special instances where all the considered tasks are accelerated when assigned to a GPU, which is the case for the sequence comparison problem addressed in this Chapter. In this special case, the time complexity reduces to $\mathcal{O}(mn \log(n))$, which is satisfactory for real implementations.

2.4 Designing the SWDUAL implementation

Our implementation is designed using the master-slave model. The master is responsible for receiving commands from the user, reading the sequences from disk, generating a list of tasks and allocating them to the workers (slaves), receiving and presenting the results back to the user. The workers first have to register themselves with the master. Then, acquire the same sequences that master received as parameters from the user, receive tasks from the master, execute them and return the results. Both the master and workers convert the format of the sequences if necessary. Figure 2.6 shows the different steps taken during execution by the master and the workers.

First, the master processes the command line arguments entered by the user. Then, it loads the sequences, converts the format if necessary and waits for the workers to connect. The workers are started either manually or automatically, connect to the master, load the sequences and also if necessary convert the format.

Now, the master can use the information gathered from the workers and the allocation policy or scheduling algorithm to allocate tasks to the workers, after which the workers start executing them. That can be done only once at the beginning of the execution or iteratively until all tasks are executed.

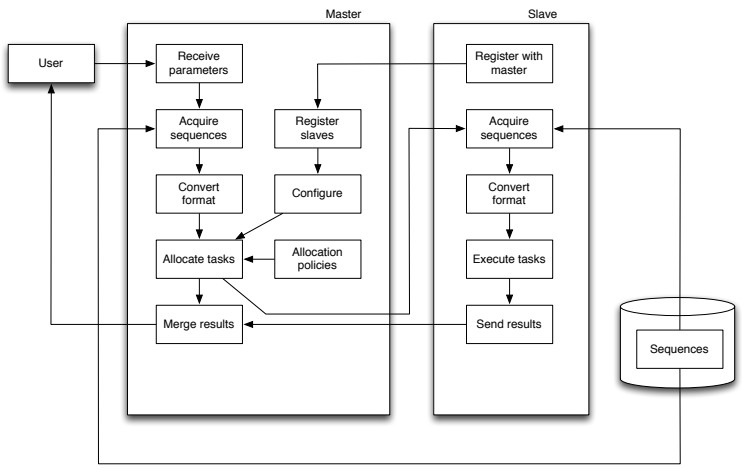


Figure 2.6 – SWDUAL master-slave model

Finally, the workers send the results to the master that merge and present them to the user.

Sequence database files created using the Fasta [54] format are in fact text files, with sequences placed one after the other. For that reason, it is not feasible to read specific sequences contained in the file, which is important for implementations like SWDUAL.

To improve this reading process, a simple binary format was created with a few additional fields. Using this format, both the master and workers are able to read sequences in any position inside the file, directly. Additionally, the memory allocation process is simplified due to the fact that the all the sequences sizes are known beforehand.

2.5 Experimental Results

In this section, the experimental results of the method implementation are presented and compared to the state-of-the-art.

The method proposed in Section 2.3 was implemented in C++ with SSE extensions and CUDA.

The strategy was implemented in C with SSE extensions and CUDA, and it integrates CUDASW++ 2.0 [44] and SWIPE [57] into the code. That code was compiled with the CUDA SDK 4.2.9 and gcc 4.5.2. The operating system used was Linux 3.0.0-15 Ubuntu 64 bits. The tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000

amino acids, which were compared to 5 real genomic databases: Uniprot with 537,505 sequences, Enbembl Dog with 25,160 sequences and Rat with 32,971 sequences and RefSeq Human with 34,705 sequences and Mouse 29,437 sequences.

The tests were executed in the Idgraf high performance computer located at Inria Grenoble. It contains 2 Intel Xeon 2.67GHz processors with 4 cores each, 74GB of RAM and 8 Nvidia Tesla C2050 GPUs. The machine was reserved for exclusive use for the duration of the test to ensure that no other major process was running concurrently. All the sequences used were available locally to minimize the influence of the network and file reading time. All combinations of programs, number of workers, query and database sequences were executed twenty-five times and the average total wall-clock execution time was recorded. Also, processor affinity was used to ensure that each process stayed in the same processor during the whole execution.

2.5.1 Comparison to other implementations

Table 2.1 shows the state-of-the-art implementations that were compared to SWDUAL, as well as their version number and command line options. For the commands, the variables were $T for the number of threads, $Q query sequence and $D database sequence.

Table 2.1 – Applications included in the comparison

Application	Version	Command line
SWIPE	1.0	./swipe -a $T -i $Q -d $D
STRIPED		./striped -T $T $Q $D
SWPS3	20080605	./swps3 -j $T $Q $D
CUDASW++	2.0	./cudasw -use_gpus $T -query $Q -db $D

The SWDUAL implementation was compared against SWIPE, STRIPED, SWPS3 and CUDASW++.

SWIPE [57] was written mostly in C++ with some parts hand coded in assembly. It was compiled using the provided Makefile.

The source code for the Farrar's STRIPED implementation of the SW algorithm [24] was compiled using the provided Makefile. It was written mainly in C with some parts also coded in assembly or Intel intrinsics.

SWPS3 [64] was downloaded from the author's website and was written in C. It was compiled using the provided Makefile.

CUDASW++ 2.0 [44] was also downloaded from the author's website and was written in C++ and CUDA. It was compiled using the provided Makefile. CUDA 4.1 was used in the compilation.

The tests were conducted using the UniProt database (*www.uniprot.org*) and 40 query sequences taken from it. Also, were used on the test up to four CPUs and four GPUs. For that reason the considered applications were executed with up to four workers, while SWDUAL, that uses both CPUs and GPUs as workers was executed with workers between two and eight. In this case, the first four workers used on the SWDUAL execution were GPUs and the last four workers were CPUs.

The reason why only four CPUs and four GPUs were used in this test although eight CPUs and eight GPUs were available is that each GPU worker actually needs some CPU time to execute as fast as it can. As a consequence, using more CPUs and GPUs than that number impacts on the overall performance of the applications and the speedup is considerably worst. Thus for the applications that only used CPUs or GPUs up to four workers were used. The exception was our case that was executed with four GPUs and four CPUs for a total of eight workers. In this case, since our implementation needs at least one CPU and one GPU to execute, we start with two workers. For three workers, two are GPUs and one is a CPU. Finally, an execution of four workers uses three GPUs and one CPU.

The SWDUAL implementation was able to significantly reduce the execution time of the sequence database searches using the Smith-Waterman algorithm compared to earlier proposals that use only one type of processing element. As can be seen on Figure 2.7 and Table 2.2, the combination of CPUs and GPUs leaded to very good results. When executing with two workers, SWDUAL showed a reduction of 54.7%, 85% and 98% when compared to the same execution on SWIPE, STRIPED and SWPS3, respectively. When executing with four workers, a reduction of 55.3% was obtained when compared to the execution on SWIPE, 73.5% when compared to STRIPED and 98.6% on SWPS3.

Also, due to the implementation of the dual approximation scheduling algorithm, the execution on each of the processing elements finished with almost no idle time.

2.5.2 Comparison to 5 genomic databases

In this case, the tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000 amino acids, which were compared to 5 real genomic databases, as in [44]: Uniprot with 537,505 sequences

Table 2.2 – Execution times for the compared implementations

Application	Number of workers			
	1	2	3	4
SWPS3	69208.2	36174.09	25206.563	18904.31
STRIPED	7190	3615.38	1369.33	1027.28
SWIPE	2367.24	1199.47	816.61	610.23
CUDASW++	785.26	445.611	350.09	292.157
SWDUAL		543.28	472.84	271.98
Application	Number of workers			
	5	6	7	8
SWDUAL	266.69	239.04	183.12	142.98

(*www.uniprot.org*), Enbembl (*www.ensembl.org*) Dog with 25,160 sequences and Rat with 32,971 sequences and RefSeq (*www.ncbi.nlm.nih.gov/RefSeq*) Human with 34,705 sequences and Mouse 29,437 sequences as listed in Table 2.3.

Table 2.3 – Genomic Databases used on the tests

Database	Number of database seqs	Smallest query seq	Longest query seq
Ensembl Dog Proteins	25,160	100	4,996
Ensembl Rat Proteins	32,971	100	4,992
RefSeq Human Proteins	34,705	100	4,981
RefSeq Mouse Proteins	29,437	100	5,000
UniProt	537,505	100	4,998

In order to measure the benefits of using a hybrid platform, the wall-clock execution time and GCUPs (billion cell updates per second) obtained were measured when comparing 40 query sequences to the five genomic databases.

As can be seen on Table 2.4, SWDUAL was able to obtain good speedups while combining CPUs and GPUs, reducing the execution time repeatedly while adding processing elements. For the Uniprot database the execution time was reduced from 543 seconds (approximately 10 minutes) to 86 sec-

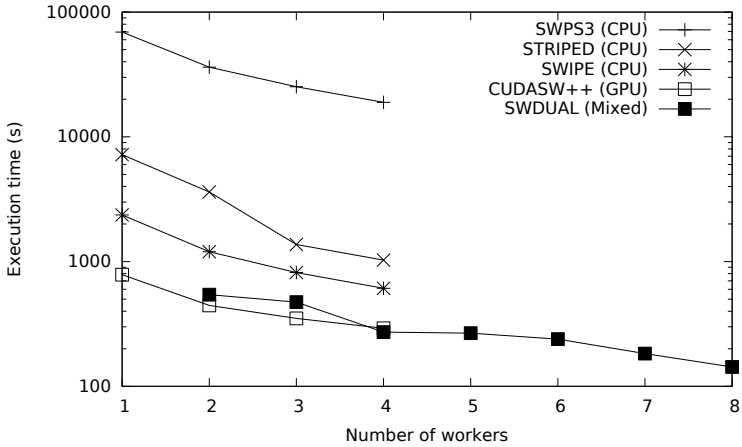


Figure 2.7 – Execution times in seconds for the compared implementations

onds when executing on eight CPUs and eight GPUs. Figure 2.8 shows the execution times obtained when comparing the databases.

Table 2.4 – Results running on GPUs and CPUs

Workers	2 Time (s) GCUPS	4 Time (s) GCUPS	8 Time (s) GCUPS
Ensembl Dog	78.36 18.91	39.63 37.39	20.45 72.45
Ensembl Rat	75.85 22.97	37.97 45.89	20.17 86.38
RefSeq Mouse	84.40 18.99	46.25 34.66	23.59 67.95
RefSeq Human	95.09 20.70	48.01 41.00	24.82 79.31
Uniprot	543.28 35.81	271.98 71.53	142.98 136.06

2.5.3 Comparison of homogeneous and heterogeneous sets

For this test, two additional query sets were created from the Uniprot database. Each query set has, like in the previous tests, 40 sequences. In this case, the sequences in the homogeneous set range in size from 4500 to

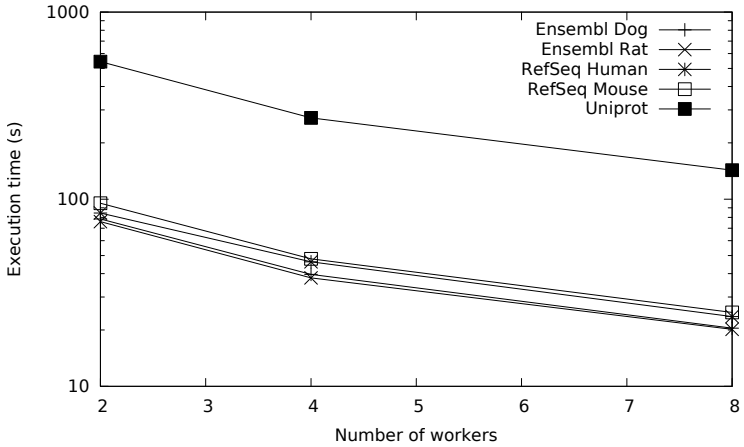


Figure 2.8 – Execution times for the compared databases

5000 and the ones in the heterogeneous set have sizes between 4 (the smallest sequence in the database) and 35213 (the largest sequence in the database).

The idea is to verify that the allocation strategy and the application as a whole is equally able to work with sequences, and therefore tasks, that are similar in terms of size as well as tasks with very different sizes.

Table 2.5 shows the execution times and the GCUPs obtained when comparing these two sets to the UniProt database. In this case, SWDUAL was able to achieve good performance on both sets. Figure 2.9 also shows the results obtained in these comparisons.

Table 2.5 – Results running the homogeneous and the heterogeneous sets

Sets	2 Time (s) GCUPS	4 Time (s) GCUPS	8 Time (s) GCUPS
Heterogeneous	3554.36 37.55	1785.73 74.74	908.45 146.92
Homogeneous	998.27 36.3	484.74 74.76	249.69 145.14

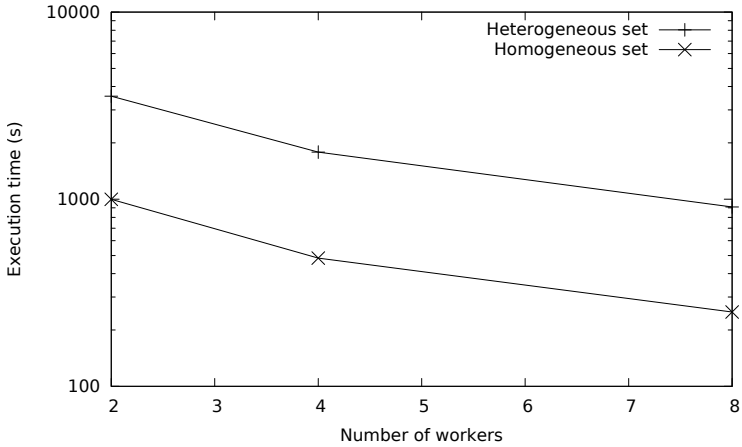


Figure 2.9 – Execution times for the heterogeneous and homogeneous sets

2.6 Conclusion

Efficient parallelization of the Smith-Waterman algorithm using SIMD and SIMT on standard hardware enables sequence database searches to be performed much faster than before. Also, scheduling algorithms like the dual approximation and the master-slave model allow for better utilization of the resources by combining the processing elements available in hybrid platforms.

In our new implementation, the comparison of a given database to a set of query sequences is divided among any number of workers. The dual approximation algorithm was used to decide which tasks to execute on the GPUs. The objective is to achieve good execution times and have as little idle time on the processing elements as possible.

When comparing 40 query sequences to the UniProt database a speed of 225 billion cell updates per second (GCUPS) was achieved on a dual Intel Xeon processor system with Nvidia Tesla GPUs, reducing the execution time from 543 seconds to 86 seconds. In addition to that, the combination of GPUs and CPUs was responsible for reducing the execution time to a total of 142 seconds for that database, which is faster than all the compared implementations.

Finally, we showed that SWDUAL is able to reduce execution times in sequence database comparisons both when tasks have similar sizes and sizes that are very different between tasks.

Chapter 3

A New Online Method for Scheduling Independent Tasks

3.1 Introduction

We are interested in studying the problem of scheduling a set of independent sequential without preemption. These tasks are submitted on a multi-core parallel machine. This is a basic scheduling problem whose different variants have been studied for different objectives. The aim is, in general, to determine good solutions in short (polynomial) time, whose objective values are close to the optimal solution.

Most existing works, and particularly theoretical studies, consider *off-line* executions, that correspond to scenarios where the entire instance is known in advance. In this case, the algorithms naturally target the minimization of objectives like the maximum completion time (*makespan*) or the total completion time of all tasks. Several provably good algorithms have been proposed for these objectives which achieve constant approximation ratios.

In this Chapter, we consider more realistic scenarios where the tasks and their characteristics are only known when they are submitted in the system. This *on-line* problem is harder since we should be able to make decisions with a partial knowledge of the instance. In this context, objectives like makespan have no concrete meaning since they could strongly depend on the maximum submission time. For these reason, in the on-line setting we usually consider objectives based on the *flow-time* which corresponds to the time that a task remains to the system. We are mainly interested in the *stretch* of the tasks (also known as *slow-down*) which normalizes the flow-time of each task with respect to its processing time.

Our purpose within this work is to introduce a new technique for on-line scheduling independent tasks which will be applied to optimize several enhanced objectives. The main idea of this technique is to detect a reasonable number of tasks that have a negative impact on the whole execution and to *redirect* them to a *dedicated pool* of processors in order to be executed apart from the other tasks. More precisely, we propose to split the set of tasks into *common* and *heavy* tasks. This split is performed on-line at run time. Informally, a task will be called heavy if its execution delays an important number of smaller tasks. Whenever a task is characterized as heavy, it is redirected to the pool of dedicated processors and this decision will never be reconsidered in the future.

The dedicated pool consists of a subset of the available processors of the machine, that is no additional processors are used. However, it will be only used by the heavy tasks. The size of this pool should be related to several parameters, including the input instance, the size of the machine, as well as, parameters that cause the redirection of the heavy tasks.

In order to characterize a task as heavy, we propose to keep track of the number of smaller tasks that arrive during the execution of a task. For this, a counter is initialized at the beginning of the execution of a task, and if this counter reaches a given threshold, then the task is characterized as heavy and it is redirected. The execution of heavy tasks on the dedicated processors is restarted from the beginning, so that the non-preemptive assumption to be satisfied. The above idea of the counter has been used in [47], where a theoretical upper bound has been proved. However, this algorithm completely rejects the heavy tasks instead of redirecting them, an action that is not permissible in real systems.

Based on the observation that most heavy tasks in the above policy have quite long processing times, we also propose a random method which characterizes a long task as heavy with some given probability. This method avoids the interruption of the tasks and the re-execution of part of them, but does not benefit of the information about the currently delayed tasks provided by the counter.

Both above ideas have been assessed by an extensive simulation campaign based on real data coming from actual execution traces. The experimental results show the significant benefits compared to standard base-line policies for scheduling tasks without preemptions.

In Section 3.2 we formally define the studied problem and we give the notations used throughout the Chapter, while in Section 3.3 we describe known results and position this Chapter in regards to related works. In Section 3.4 we first present the base-line non-preemptive algorithms as well as

the preemptive lower bounds that will be used in our experiments. Then, we propose our algorithms based on heavy tasks and redirections. In Section 3.5 we describe the construction of experimental data which are extracted from 7 different traces. Next, we provide the results of the experimental campaign consisting of the parameter tuning of our new methods as well as their comparison in the constructed instances with the base-line algorithms and the lower bounds. Finally, we conclude in Section 3.6.

3.2 Definitions and notations.

In the following, we consider a set of n independent sequential tasks and a parallel machine consisting of m processors. Each task should be executed on a single processor without *preemptions*. We denote by M_i, $1 \leq i \leq m$, each processor of this machine. Each task T_j, $1 \leq j \leq n$, is characterized by a *processing time* p_j, which becomes known to the scheduler only after its submission at time r_j; we call r_j the *release time* of T_j. In what follows we denote by $p_j(t)$ the remaining processing time of the task T_j at time t. Note that $p_j(r_j) = p_j$.

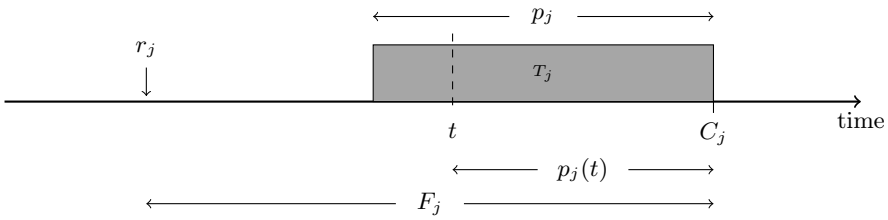


Figure 3.1 – Summary of notations.

Given a schedule, C_j denotes the *completion time* of T_j, that is the time at which it is completed. From a more realistic perspective and taking into account the user point of view, a natural measure of the quality of service delivered to a task is the amount of time it spends in a system. The basic objective in this direction is the *flow-time* (also called response time) of a task, which is defined as the amount of time it remains in the system until being completed. In other words, the flow-time of a task is composed by the time this task waits in the system to begin its execution plus its processing time. The flow-time of a task T_j is denoted by $F_j = C_j - r_j$.

Many variants of flow-time metric have been investigated in different settings. Since the flow-time depends on the processing time, it varies a lot for different tasks. For this reason the *stretch* (or slowdown) metric has

been proposed which normalizes the flow-time of a task by dividing it by its processing time, *i.e.* $S_j = {}^{F_j}/_{p_j}$. This metric is often used in fair scheduling where the users are willing to wait longer for long tasks as opposed to short ones. However, the stretch metric has also some drawbacks since it largely focus on very short tasks. Specifically, such a short task cannot wait even for a natural period of time because its stretch will be exploded. To overcome this, the stretch metric has been refined to the *bounded stretch* (or bounded slowdown) metric [26] which is defined as $\frac{F_j}{\max\{p_j,B\}}$, where $B > 1$ is a small constant (we set $B = 10$). In this Chapter we are mainly interested in the bounded stretch metric, and secondarily in flow-time. For simplicity, we will henceforth refer to bounded stretch by just stretch.

For both stretch and flow-time, we consider three different objective functions. The first objective function concerns the minimization of the maximum stretch or flow-time over all tasks. These objectives are denoted by $S_{\max}$ and $F_{\max}$, respectively. The second objective function corresponds to the minimization of the total stretch or flow-time for all tasks, *i.e.* $\sum S_j$ and $\sum F_j$, respectively. A third objective function that balances the previous two functions, while it is also considered to increase the fairness among tasks, is the norm. We will consider the second norms of stretch and flow-time which are defined as $(\sum S_j^2)^{1/2}$ and $(\sum F_j^2)^{1/2}$, respectively. Note that the first norm corresponds to the sum function, while the infinite norm to the max function.

3.3 Related works

Scheduling efficiently concurrent tasks on a parallel machine is a central problem for reaching good performances. There exist a huge literature on this problem focusing on standard objectives based on completion times (see for example [22]). Most of these studies are done in an off-line setting. Here, we are interested in flow-time based objectives that are more appropriate for the non-preemptive and mainly for the on-line settings. In general, the flow-oriented objectives are harder than the standard makespan or total completion time objectives.

In what follows, we say that an off-line algorithm achieves a ρ-approximation ratio if the objective value of its constructed solution is at most ρ times bigger than the objective value of an optimal solution. In a similar way, we may define the competitive ratio for on-line algorithms, comparing the algorithm's solution with the off-line optimal solution.

If preemptions are allowed, the classical Shortest Remaining Processing

Time policy (SRPT) provides an optimal solution for minimizing the total flow-time and a 2-competitive solution for minimizing the total stretch on a single processor [12]. Moreover, a variant of SRPT achieves a competitive ratio of 13 for the total stretch minimization problem on parallel processors [52]. Furthermore, Bender *et al.* [13] proposed a variant of the classical Earliest Deadline First (EDF) policy for the max stretch objective on a single processor which leads to a $\mathcal{O}(\sqrt{\Delta})$-competitive algorithm, where Δ is the ratio of the maximum over the minimum processing time of the instance. As we consider non-preemptive scheduling, the above results may only serve as lower bounds.

In the non-preemptive case, Kellerer *et al.* [39] showed that there exists a lower bound of $\Omega(n^{\frac{1}{2}-\epsilon})$ on the approximability of total flow-time minimization problem even on a single processor, while the corresponding lower bound for the on-line case is $\Omega(n)$ [17]. On the positive side, an $\mathcal{O}(\sqrt{\frac{n}{m}}\log\frac{n}{m})$-approximation algorithm has been presented in [41] for the problem of minimizing the total flow-time on m parallel processors. Moreover, Weighted Shortest Processing Time (WSPT) is a $\mathcal{O}(\Delta + \frac{3}{2} - \frac{1}{2m})$-competitive algorithm [65] for the more general problem where each task has a weight and the objective is to minimize the total weighted flow-time. Since this problem is a generalization of the total stretch minimization problem, to which reduces if the weight of a task is equal to the inverse of its processing time, the result holds also for the latter objective.

Concerning the max function, it is known that First-Come-First-Served (FCFS) is optimal on a single processor and $(3 - \frac{2}{m})$-competitive on parallel processors for the max flow-time minimization problem [13]. Bender *et al.* [13] showed that the max stretch minimization problem on parallel processors cannot be approximated within a factor of $\Omega(\Delta^{\frac{1}{3}})$ on a single processor and $\Omega(n^{1-\epsilon})$ on parallel processors, unless $\mathcal{P} = \mathcal{NP}$. Legrand *et al.* [40] studied the FCFS policy for the problem of minimizing the max stretch on a single processor and they proved that it achieves a $\mathcal{O}(\Delta)$-competitive ratio. This result has been improved to $(\frac{\sqrt{5}-1}{2}\Delta + 1)$ in [23]. No results are known for the max stretch objective in multi-processors setting.

Let us conclude this section by a remark: the idea presented in this Chapter is related to the over-provisioning mechanism, which is for instance discussed in [58] in the context of energy saving in high-performance parallel platforms. The idea is to add extra hardware (processors) in order to improve the power utilization as well as the task throughput in power-constrained platforms. In our case, we propose to dedicate a part of the processors to a specific utilization on a small number of problematic tasks for improving

the whole execution of the bunch of tasks.

3.4 Algorithms

In this section we propose new policies for scheduling sequential tasks on parallel processors based on the idea of redirecting the "hard" tasks to a dedicated subset of processors. Before this, we briefly describe several standard scheduling policies whose efficiency is compared in Section 3.5 with the efficiency of our algorithms.

In all cases, we will consider two basic paradigms: the *common queue* and the *immediate dispatch* models. In the first model, we use a single common queue Q for all processors in order to store the *pending* tasks, i.e., the tasks that are released but not yet executed. Then, whenever a processor becomes idle we select a task from Q according to rules that depend on the specific policy. In the second model, we use a different queue Q_i for each processor M_i and we dispatch each task just upon its arrival to one of these queues. We will describe the *dispatching policy* in Section 3.4.4, after presenting the scheduling policies.

3.4.1 Standard scheduling policies

Initially, we describe the policies based on a common queue Q for all processors. Specifically, we will consider the following three "global" policies.

First-Come First-Served (FCFS) Whenever a processor becomes idle, schedule on it the earliest released task in Q.

Global Shortest Processing Time (GSPT) Whenever a processor becomes idle, schedule on it the shortest processing time task in Q.

Global Shortest Remaining Processing Time (GSRPT) Whenever a processor becomes idle, schedule on it the shortest remaining processing time task in Q. At the arrival of a new task T_j we search for the processor that actually executes the task with the longest remaining processing time. Let M_i be this processor and T_k be the task executed on M_i at r_j. If $p_j \leq p_k(r_j)$ then we interrupt the execution of T_k and we start executing T_j on processor M_i, while the task T_k is added again to Q with processing time $p_k(r_j)$.

Next, we describe the immediate dispatch versions of the SPT and SRPT policies, respectively. Recall that the dispatching policy will be presented in Section 3.4.4.

Shortest Processing Time (SPT) Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i.

Shortest Remaining Processing Time (SRPT) Whenever a processor M_i becomes idle, schedule on it the shortest remaining processing time task in Q_i. At the arrival of a new task T_j which is dispatched to processor M_i, let T_k be the task which is actually executed by M_i. If $p_j \leq p_k(r_j)$ then we interrupt the execution of T_k and we start executing T_j on processor M_i, while the task T_k is added again to Q_i with processing time $p_k(r_j)$.

Note that the policies based on SRPT produce preemptive schedules, and they are used as lower bounds. Moreover, all the above policies apart from FCFS prioritize the tasks with the shortest (remaining) processing time. The motivation to this is due to the fact that this order provides good schedules for both the total flow-time and the total stretch objectives. Recall that, for the single processor case, SRPT is a preemptive policy which is optimal for total flow-time [12] and almost optimal for total stretch, while SPT is an optimal non-preemptive policy for both objectives if there are no release dates [63]. On the other hand, it is well-known that FCFS is an optimal non-preemptive policy for the maximum flow-time objective on a single processor.

3.4.2 Scheduling with redirections

In this section we propose new algorithms for scheduling non-preemptively a set of sequential tasks on a multi-processor machine. The basic idea is to detect the *heavy* tasks whose non-preemptive execution increases significantly the waiting time of smaller tasks. These heavy tasks will be then *redirected* to be executed on a small subset of processors reserved exclusively for them.

In order to give the intuition of our idea, let us present an example that shows why any on-line non-preemptive policy $\mathcal{A}$ can be arbitrarily bad with respect to an optimal non-preemptive schedule. For simplicity, we consider that a single processor is available. Assume that a big task T_0 of processing time p is released and, without loss of generality, the scheduler starts executing T_0 at time 0. Then, p unit processing time tasks are released: task T_i, $1 \leq i \leq p$, is released at time $r_i = i$. Since $\mathcal{A}$ is a non-preemptive algorithm, it cannot interrupt the big task and hence executes the small tasks during the interval $[p+1, 2p+1]$ as, for example, shown in Figure 3.2. The total

flow-time of any such schedule is $O(p^2)$. On the other hand, the offline optimal non-preemptive schedule will delay the execution of the big task and execute all small tasks before T_0, which gives a schedule of total flow-time only $O(p)$.

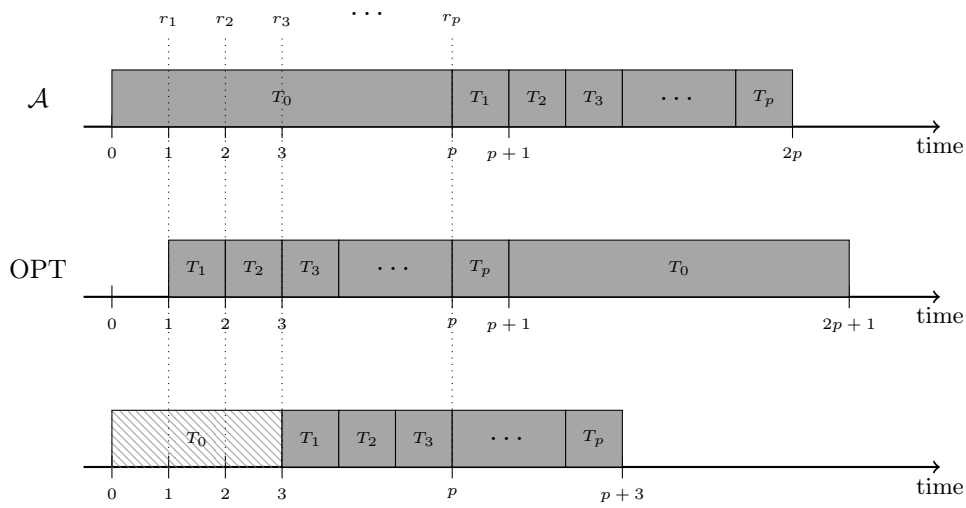


Figure 3.2 – Intuition of redirection.

Motivated by this example, we propose to characterize a task T_j as *heavy* if the number of smaller tasks that arrive during its execution attains a given threshold τ. When this threshold is attained, the execution of T_j is interrupted and it is redirected to a small subset of processors dedicated only for heavy tasks. Note that, the execution of any heavy task on these processors will be restarted from the beginning, that is heavy tasks are not preemptively executed but they are interrupted and restarted. The last part of Figure 3.2 demonstrates an example of the resulting schedule if the threshold is set to be $\tau = 3$. In this case, the task T_0 is interrupted at time $r_3 = 3$ while the remaining tasks can start their execution and complete earlier than in the schedule of $\mathcal{A}$ but later than in the optimal schedule. In this way, the value of the threshold could describe a trade-off between the number of the tasks that should be restarted and the performance for the remaining tasks.

As in the case of the standard scheduling policies, we will consider two versions of our method, one with a common queue and one with immediate dispatch. The formal definition of these two versions follows. In both cases, we consider that a threshold τ is given.

Global Shortest Processing Time with Redirections (GSPT-RD) Whenever a processor becomes idle, schedule on it the shortest processing time task in Q. At the beginning of the execution of a task T_k on any processor M_i we introduce a counter c_k which is initialized to zero. At the arrival of a new task T_j, we search for the processor that actually executes the task with the longest remaining processing time. Let M_i be this processor and T_k be the task executed on M_i at r_j. If $p_j < p_k(r_j)$ then we increase by one the counter c_k. If $c_k = \tau$ then we interrupt T_k and we redirect it to the set of dedicated processors.

Shortest Processing Time with Redirections (SPT-RD) Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i. At the beginning of the execution of a task T_k on any processor M_i we introduce a counter c_k which is initialized to zero. At the arrival of a new task T_j which is dispatched to processor M_i during the execution of T_k, we increase by one the counter c_k only if $p_j < p_k(r_j)$. If $c_k = \tau$ then we interrupt T_k and we redirect it to the set of dedicated processors.

It remains to describe the scheduling policy that we use for the subset of the processors dedicated for the redirected tasks. In fact, in each case we use the same policy as for the main set of processors, without however allowing new redirections for the already redirected tasks. In other words, the GSPT (resp., SPT) policy is used for the dedicated processors along with the GSPT-RD (resp., SPT-RD) policy.

3.4.3 Scheduling with random redirections

In some applications, the interruption of a task is very complicated even if it will be re-executed from the beginning. For example, this is the case of output-intensive applications. For this kind of applications, we propose to substitute the threshold rule used in the previous section with a random rule which will be applied before the beginning of the execution of each task. In this way, no interruptions will be performed.

In order to simulate the threshold rule, we observe that the tasks redirected based on it are in general big tasks. For this reason, we characterize a task as *potentially heavy* if its processing time is at least $b \cdot p_{\max}$, where $b \in [0, 1]$ is a constant that characterizes how big is the task and $p_{\max}$ is the maximum processing time over all tasks that have been arrived by the current time. Then, a potentially heavy task will be redirected with a probability $c \in [0, 1]$. Note that a significant drawback of the random rule with respect to the threshold rule is that the first one only cares about the load

of the task under consideration, while the second one relates the load of this task with the current load of the system.

We again consider two versions of the random method which are formally defined as follows.

Global Shortest Processing Time with Random Redirections (GSPT-RR) Whenever a processor becomes idle, schedule on it the shortest processing time task in Q. At the arrival of a new task T_j, if $p_j \geq b \cdot p_{\max}$ then select a number c_j uniformly at random in $[0, 1]$. If $c_j \leq c$ then redirect T_j to the set of dedicated processors. In any other case, add T_j in Q.

Shortest Processing Time with Redirections (SPT-RR) Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i. At the arrival of a new task T_j, if $p_j \geq b \cdot p_{\max}$ then select a number c_j uniformly at random in $[0, 1]$. If $c_j \leq c$ then redirect T_j to the set of dedicated processors. In any other case, apply the dispatching policy for the task T_j.

3.4.4 Dispatching policy

In order to decide to which processor we will dispatch each new task, we use a rule that tries to balance the flow-times of tasks over all processors. Specifically, for each processor separately, we compute the *marginal increase* in the total flow-time that causes the potential assignment of the new task to this processor. Then, we dispatch the new task to the processor for which this increase is minimum. Note that this decision is irrevocable, except for the redirection case, but still the processors used for redirected tasks are apart.

In what follows in this section, we clearly define the marginal increase in different cases. Observe first that in any of the presented scheduling policies which are based on the immediate dispatch model (SPT, SRPT and SPT-RD), the tasks are executed in shortest (remaining) processing time order. Based on this observation, we can say that the marginal increase that causes a new task T_j if it is dispatched to a processor M_i consists of the flow-time of T_j as well as the increase of the flow-time for the tasks of processing time bigger than p_j in the queue Q_i.

Assuming that the task T_k is executed at r_j on M_i, the marginal increase can be defined in the general case as follows:

$$\Delta\mathcal{F}_i = \left(p_k(r_j) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) \leq p_j}} p_\ell(r_j) + p_j\right) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) > p_j}} p_j$$

where the first part is the potential flow-time of T_j if it is dispatched to M_i, while the second part corresponds to the potential delay for the tasks of longer (remaining) processing time than T_j. Note that, in the cases of SPT and SPT-RD, for the value of $p_\ell(r_j)$ in the first sum, it always holds that $p_\ell(r_j) = p_\ell$.

The above general definition has two exceptions. First, in the case of SRPT and only if the potential dispatching of the new task T_j on M_i will preempt the executed task T_k, i.e., if $p_j < p_k(r_j)$, then the marginal increase is defined as follows:

$$\Delta\mathcal{F}_i = p_j + \sum_{\substack{T_\ell \in Q_i \cup \{T_k\}: \\ p_\ell(r_j) > p_j}} p_j$$

Second, in the case of SPT-RD and only if the potential dispatching of the new task T_j on M_i will redirect the executed task T_k, i.e., if $p_j < p_k(r_j)$ and $c_k = \tau - 1$, then the marginal increase is defined as follows:

$$\Delta\mathcal{F}_i = \left(\sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) \leq p_j}} p_\ell(r_j) + p_j\right) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) > p_j}} p_j - \sum_{T_\ell \in Q_i} p_k(r_j)$$

where the negative term corresponds to the decrease in the flow-time of all tasks in Q_i by the removal of the remaining processing time of T_k.

3.5 Experimental results

In this section we describe the generation of the instances that we will be next used to experimentally compare our new methods based on redirections with the state-of-the-art algorithms for scheduling independent tasks.

3.5.1 Construction of instances

In our experiments we use data from 7 traces in order to validate the results across different scenarios. The traces used and their characteristics are shown in the first part of Table 3.1. Since these traces consist of parallel independent jobs, we next describe how to meaningfully transform them to construct instances of sequential independent tasks.

For each trace we create 20 instances. Specifically, in order to create an instance we uniformly at random select an initial point t during the whole time horizon of the corresponding trace. Then, all jobs that are released in the interval $[t, t+s]$ will belong to the instance, where s is the instance size (total duration) as this is given for the different traces in the last column of Table 3.1. Let n be the number of all selected jobs for an instance and m be the total number of processors used by the platform in the corresponding original trace.

Trace	Traces characteristics			Instances characteristics		
	# CPUs	# tasks	size	# CPUs	# tasks	size
Curie [1]	80640	279991	6089	160	70215	60
HPC2N [3]	240	201998	1268	17	4516	30
KTH-SP2 [10]	100	20483	333	12	860	15
LLNL-Thunder [4]	4096	97875	148	48	4601	7
Metacentrum [6]	806	86586	157	18	4155	7
RICC [7]	8192	431547	153	200	19401	7
SDSC-Blue-4.1 [8]	1152	195587	979	27	6263	30

Table 3.1 – Traces and instances characteristics. The processors and tasks numbers are the average over the 20 instances created for each trace.

Each job is characterized by a *real execution time* e_j, a *number of required processors* q_j and a *release date* r_j. We consider only the real execution times since user submitted execution times are strongly overestimated. As an example, one third of the jobs in the *Curie* trace have user submitted execution times of 24 hours which corresponds to the default setting of the system. In order to transform a parallel job to a sequential task, we used the following three methods. Note that, the main characteristic of all of them is that they keep the ratio of the total execution volume over the number of processors for the created instance the same as in the initial trace.

A. For each parallel job, create q_j sequential tasks each one of processing time e_j.

B. For each parallel job, create a sequential task with processing time $p_j = q_j \cdot e_j$.

C. Let $Q = (\sum_{j=1}^{n} q_j)/n$ be the average number of requested processors for all jobs of the instance. For each parallel job, create a sequential task with processing time $p_j = \frac{q_j}{Q} \cdot e_j$. Moreover, the number of processors that will be used for this instance is m/Q.

We have performed some initial experiments for these three methods. However, methods A and B have significant drawbacks. Method A generates many tasks, and it can create instances that are too big to be kept in memory. Moreover, it constructs instances where multiple tasks have the same processing times. Method B overcomes both previous drawbacks. Although it creates a single sequential task per parallel job with the same execution volume, it disturbs the relations between the execution times of jobs. For example, in an instance created by method B, one job of short execution time and big number of required processors and another job of long execution time and small number of required processors could lead to tasks of the same processing times, or even worse change their ordering with respect to their execution times. For this reason, in the experimental campaign that we present in the following section, the instances are created using method C. The only (small) disadvantage of this method is that the number of processors in each instance can be significantly smaller than the number of processors used by the platform in the corresponding original trace. However, this could be considered as a natural assumption when dealing with sequential tasks.

3.5.2 Experiments

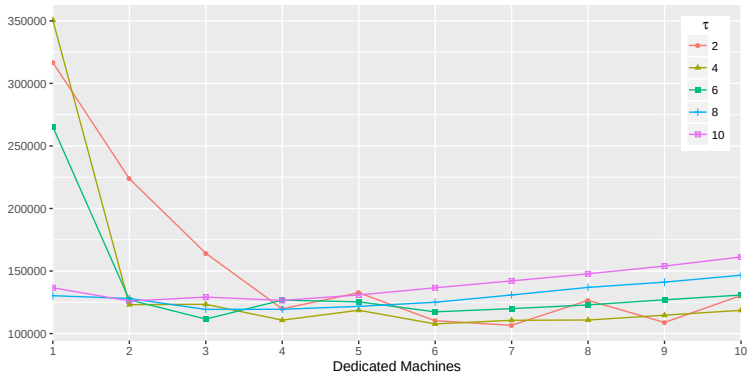
All the algorithms have been implemented in a custom discrete event simulator. The code of the simulator and the experimental data used can be found at `http://fernando.mendonca.xyz/redirection.tar.gz`. All values given in the following subsections and figures correspond to the average over the 20 instances constructed as described above for each trace.

Tuning of parameters The first part of our experiment concerns the tuning of the parameters used for methods SPT-RD, GSPT-RD, SPT-RR and GSPT-RR. Specifically, for the methods SPT-RD and GSPT-RD we have to define the number of dedicated processors m_d used for the heavy tasks and the threshold τ which determines the characterization of a task as heavy. Recall that the dedicated processors is a subset of the processors of the machine and hence $m_d < m$. In other words, the basic set of processors used for the common tasks contains $m - m_d$ processors.

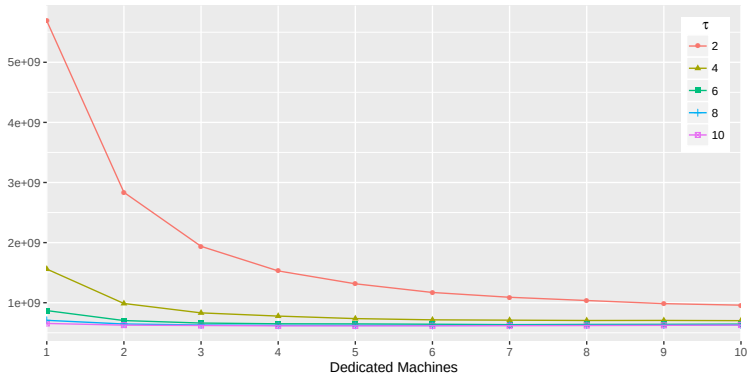
For *Curie* and *RICC* traces we will set $\tau \in \{1, 2, \ldots, 10\}$, while for the remaining instances we set $\tau \in \{1, 2, \ldots, 5\}$. Moreover, we use $m_d \in \{1, 2, \ldots, 10\}$ and test all combinations of parameters.

In Figures 3.3 to 3.6, the values of total stretch and total flow-time of SPT-RD for different parameters are shown. We present here only the two

more representative traces: *Curie* and *LLNL-Thunder*. In the *Curie* trace, we observe that while the stretch is, in general, increasing with the number of dedicated processors for $m_d \geq 4$, the flow-time decreases. This is not true for the *LLNL-Thunder* trace, but a similar situation is observed for this trace with the threshold increase (see the following couple of graphs in Figures 3.3 to 3.6). In the four first graphs, the horizontal axis corresponds to the number of dedicated processors m_d while each line corresponds to a threshold τ. The inverse situation appears in the four latter graphs. In general, we have observed a very variant relation between the two parameters and the different objectives. The value of the threshold directly affects the number of redirected tasks, while indirectly may affect the under-/over-utilization of the dedicated or the basic set of processors. For this reason, the two parameters are strongly related and they should be selected in conjunction.

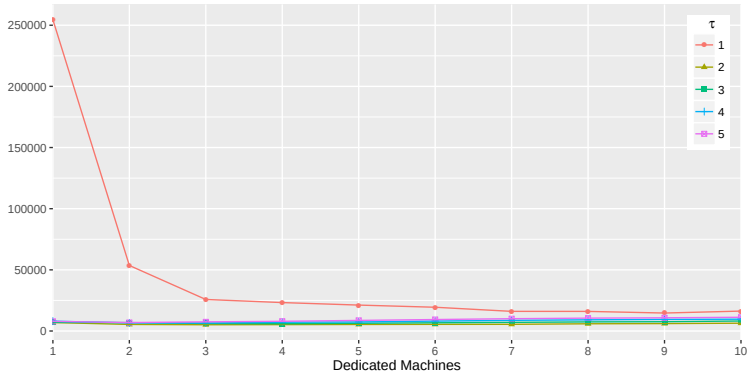


(a) Curie – total stretch

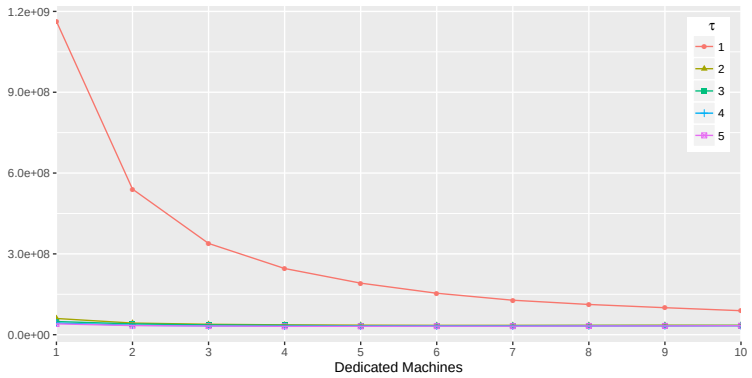


(b) Curie – total flow-time

Figure 3.3 – Total stretch and total flow-time for the *Curie* trace



(a) LLNL-Thunder – total stretch

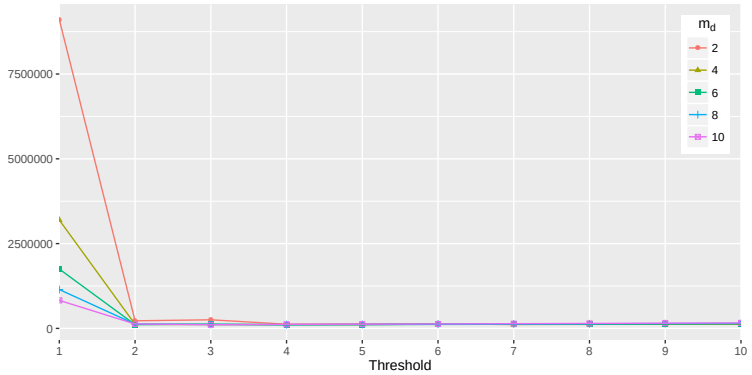


(b) LLNL-Thunder – total flow-time

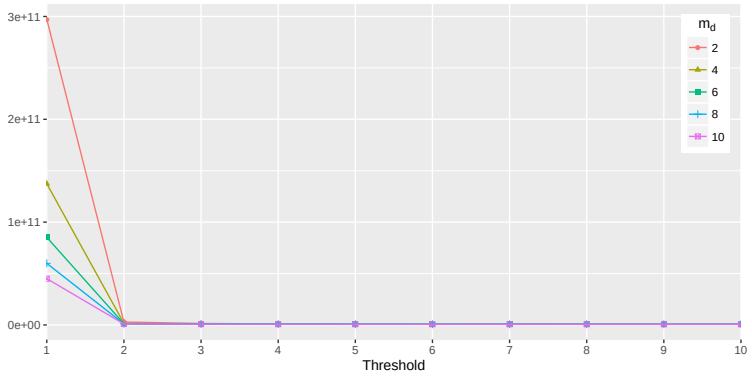
Figure 3.4 – Total stretch and total flow-time for the *LLNL-Thunder* trace

Similar observations can be done for SPT-RR and GSPT-RR methods. Here we have three parameters: the number of dedicated processors m_d, the parameter b that determines how long is a task and the probability of redirection c. In order to deal with the parameter b, we first observed that all traces have a very small number of very long tasks. To see this, we calculated the ratio of the $0.8n$ longest task of each trace over the longest task of it, and this is equal to 0.02 in average for all traces. For this reason, we fix $b = 0.02$. For the other two parameters we did a similar analysis using the values $m_d \in \{1, 2, \ldots, 10\}$ and $c = \{0.2, 0.4, 0.6, 0.8, 1\}$.

In Table 3.2, we present the "best" combination of parameters for each trace and method which will be also used in the next section. These parameters are chosen in such a way that each method has a good (or average) performance for all objectives. Another choice could be the parameters that



(a) Curie – total stretch



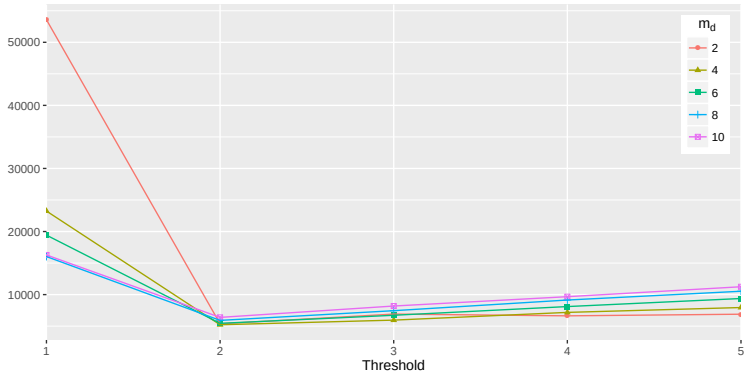
(b) Curie – total flow-time

Figure 3.5 – Total stretch and total flow-time for the *Curie* trace

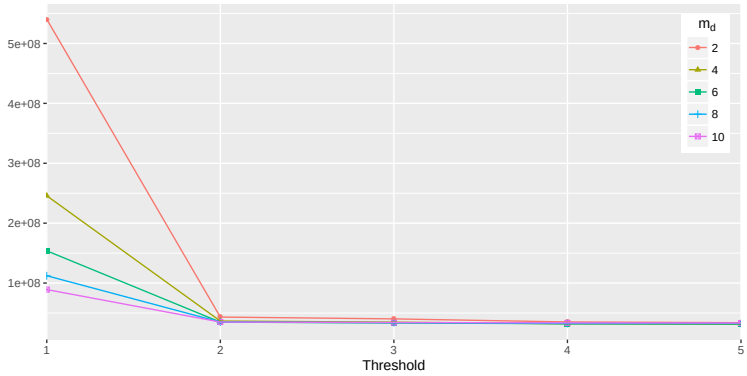
minimize a particular objective. However, this choice can have the opposite effect for the other objectives as it is indicated in the previous Figures.

Comparison with base-line algorithms and lower bounds The second part of our experiment is the comparison between the implemented methods SPT, GSPT, SRPT, GSRPT, SPT-RD, GSPT-RD, SPT-RR and GSPT-RR.

We start by focusing on the *Curie* trace which is the largest trace of our collection. The comparison of the algorithms for the different metrics can be found in Figures 3.7a to 3.7f. These Figures show that although SPT is stable for the flow metric, it is highly detrimental to the stretch metric with regard to the lower bounds (SRPT and GSRPT). On the other hand, the redirection



(a) LLNL-Thunder – total stretch



(b) LLNL-Thunder – total flow-time

Figure 3.6 – Total stretch and total flow-time for the *LLNL-Thunder* trace

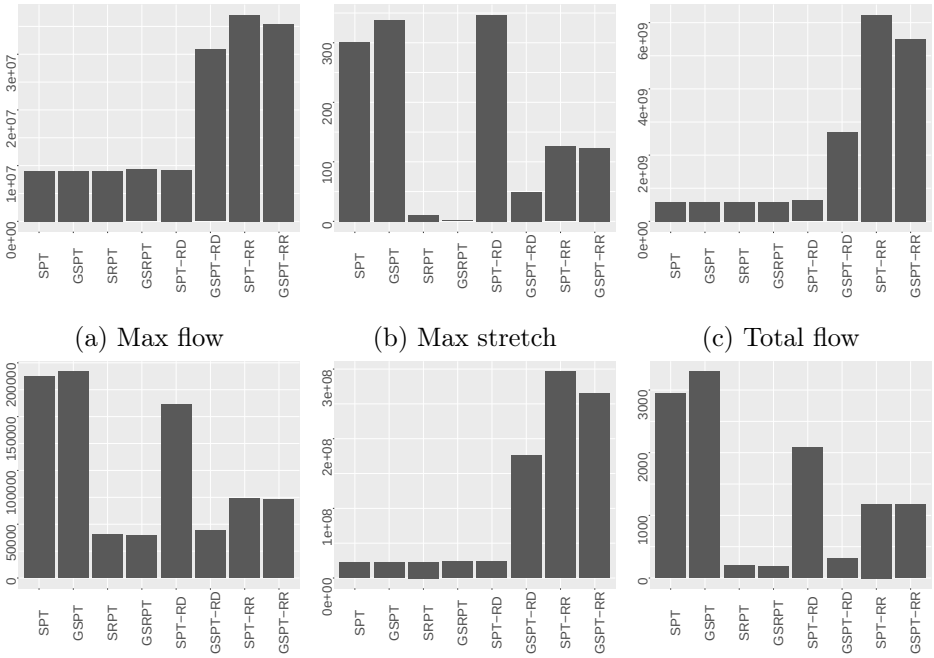
of tasks improve the stretch metrics for the *Curie* trace. Specifically, there is a significant decrease in all stretch metrics for the GSPT-RJ, SPT-RR and GSPT-RR methods. The reason that the same improvement cannot be seen for the SPT-RJ method is the presence of bursts of short tasks in the *Curie* trace. These bursts cause the redirection even of short tasks to the dedicated processors and their consequent over-utilization. For this reason, we can safely say that the flow metric would see significant improvement if more dedicated processors were used. This can be also justified by the tendency shown in Figure 3.4b.

Next, we consider the remaining traces using only the stretch metric as can be seen in Figures 3.7 to 3.13. These Figures show that in most of the cases there is a significant improvement in all stretch metrics if redirection is

Trace	SPT-RD		GSPT-RD		SPT-RR		GSPT-RR	
	m_d	τ	m_d	τ	m_d	c	m_d	c
Curie	10	10	10	4	9	0.8	10	0.8
HPC2N	5	2	10	1	5	1	10	1
KTH-SP2	8	1	7	1	2	0.8	3	0.8
LLNL-Thunder	7	2	10	5	8	1	9	1
Metacentrum	4	5	3	4	3	0.4	2	0.2
RICC	10	5	10	10	5	0.2	5	0.2
SDSC-Blue-4.1	5	2	10	1	10	1	8	0.8

Table 3.2 – Selected parameters for each trace

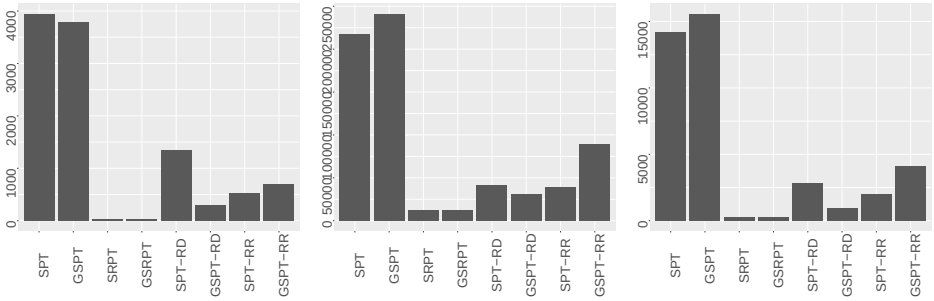
used comparing to the SPT and GSPT methods. In addition, we can see that in some cases the stretch metric for the methods that employ redirections is even close to the SRPT and GSRPT methods, justifying the efficiency of our methods.



(a) Max flow (b) Max stretch (c) Total flow

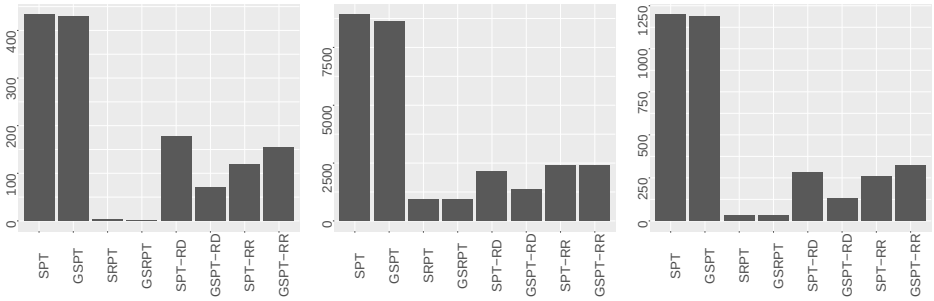
(d) Total stretch (e) Flow norm (f) Stretch norm

Figure 3.7 – Curie



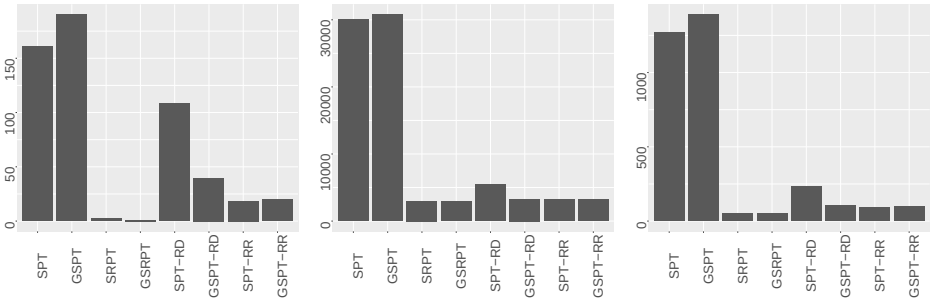
(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.8 – HPC2N



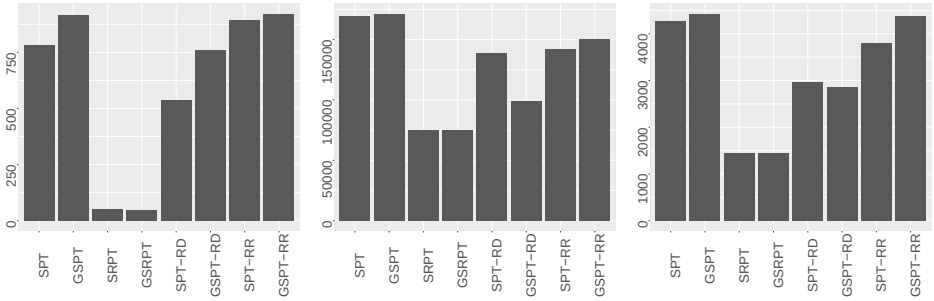
(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.9 – KTH-SP2



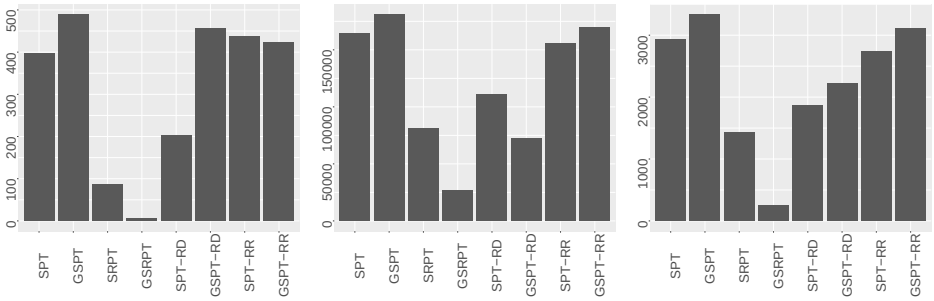
(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.10 – LLNL-Thunder



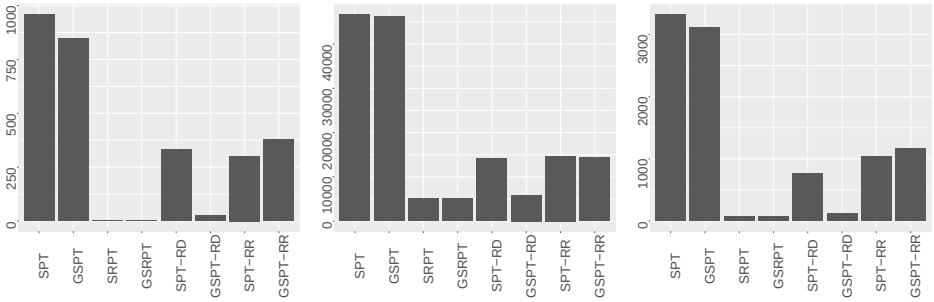
(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.11 – Metacentrum



(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.12 – RICC



(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 3.13 – SDSC-Blue-4.1

3.6 Conclusion

We have presented a method for on-line scheduling independent tasks on a multi-processor machine based on the characterization of a small number of tasks as heavy, that is problematic for the whole performance of the system. These tasks are executed on a dedicated subset of processors. We proposed a deterministic and a random approach for detecting the heavy tasks. These methods are evaluated on instances extracted by seven real traces. The experimental results show the dependency of both methods on the correct choice of the parameters. Both methods focus mainly on stretch metrics at the expense of flow-time metrics. In general, the deterministic method outperforms the standard SPT policy and in many cases its performance approached the performance of the preemptive policy SRPT which can be considered as lower bound. The gains for the random method are smaller but, in several cases, significant comparing again with SPT.

The most intriguing future direction is to extend the above ideas for parallel jobs. However, there are several issues that should be considered. First, the size of the dedicated processors part should be large enough for dealing with wide jobs, which could lead to under-utilization of these processors. A solution to this could be to leave the borders between dedicated and normal processors more free. Second, the counter method is not anymore clear how it will be applied since a new job may need to cause the redirection of more than one running jobs. Third, the commonly used backfilling strategy should also be able to benefit from redirections. In other words, redirection could be used to increase the available space (holes) in order to backfill more efficiently.

Chapter 4

Contiguity and Locality in Backfilling Scheduling

4.1 Introduction

High Performance Computing systems are evolving from large scale multi-cores to extreme scale heterogeneous many-cores (where hundreds of cores are integrated within the same chip) [21]. The number of cores will drastically increase but the I/O and interconnection devices are evolving much slowly while the memory hierarchy will be even deeper than today. In addition, more processing capabilities will obviously lead to applications with more data produced, stressing even more the interconnections.

Managing the resources in HPC platforms becomes the crucial issue. The basic scheme used is the following: users are submitting their jobs in a waiting queue while the *job and resource manager* collects data of ready jobs and analyzes them. Finally, it determines an allocation according to the available resources. The most popular scheduling mechanism in supercomputing centers is First Come First Served (FCFS). It consists in executing the jobs in their order of arrival and to allocate them in the first available time slot. FCFS is used with local improvements aiming at filling idle times with smaller jobs in the queue (Backfilling) [25]. It is well-known that such a strategy does not optimize any sophisticated function, however, it provides good results in particular because the resource occupation is high, it is simple to implement and it guarantees that there is no starvation (i.e., every job will be scheduled).

There are currently several implementations of such management systems. SLURM (Simple Linux Utility for Resource Management) provides the

infrastructure for basic allocations/monitoring/scheduling operations [66]. It is open source and it is widely used, available on more than half of the TOP500 platforms. The other management systems are special purpose systems developed by private companies (Maui, Moab, Torque) [5].

Although batch scheduler systems exist for many years, the current revolution of extreme scale computing makes them more complex because of the scalability and the heterogeneity of resources. Taking into account topological constraints is a way of handling their complexity. It is also worth in the direction of reducing the energy/power consumption [21]. With platforms using several levels of hierarchy, the difference between communication latency and bandwidth in the same level and across different levels is considerable. For this reason, it is crucial to analyze how jobs are allocated on the available processors and how this process impacts the overall system performances.

Our goal within this work is to show that some simple topological constraints are worth to be considered. Two major constraints are studied, namely *contiguity* and *locality*. The contributions of this Chapter are twofold. First, we propose a systematic theoretical study for bounding the effects of constraints on the makespan for all possible scenarios. The main result here is that the worst case ratios are all bounded by a small constant. Second, we provide an extensive experimental campaign based on actual logs. The experiments are based on the makespan metric. Experiments show good performance for the proposed algorithms with the benefit of increased jobs locality. In particular they demonstrate that contiguity is able to achieve locality without knowledge of the platform. We also notice that the proposed algorithms can easily be integrated into existing batch schedulers (i.e., with only few modifications), for instance as plugins of SLURM.

The Chapter is organized as follows. In Section 4.2 we discuss the most relevant related works concerning the scheduling problem under topological constraints. The problem description including our main hypothesis are presented in Section 4.3. Section 4.4 provides the theoretical analysis of the degradation ratio due to the additional topology-aware constraints. Various allocation algorithms targeting locality are described in Section 4.5. Finally, we report the results of our experiments in Section 4.6 and we conclude in Section 4.7.

4.2 Related Works

Backfilling algorithms are the most commonly used scheduling algorithms in batch schedulers [32, 61]. All Backfilling algorithms work by assigning jobs in a queue by incoming order to computing resources.

Conservative Backfilling is the most commonly version mentioned in the literature [25, 27]. In this version, backfilling is done only if it does not delay *any* of the previous jobs in the queue. Users provide each submitted job with a running time limit and a fixed number of processors. This knowledge allows the scheduler to know in advance all resources consumption and to backfill jobs into inactivity periods. The main feature is that no starvation occurs in conservative Backfilling.

Pascual et al. proposed *topology-aware* backfilling techniques in [53]. They introduced modifications to processors allocation algorithms in order to improve job locality in k-ary n-trees platforms topologies. They show an increase in waiting times for strict locality enforcing. While we use a similar kind of techniques, modifying the processors allocation of the Backfilling algorithm, we focus here on different constraints. Our constraints do not rely on a complex platform description. In particular scheduling with contiguous sets of processors does not require knowledge of the platform and is still subject to locality improvement. Simpler constraints also allow us to achieve a theoretical analysis of the impact of additional constraints on the makespan.

On the theoretical side, Bładek et al. studied in [15] the impact of contiguity on makespan. They provide an algorithm converting any non-contiguous schedule to a contiguous schedule with performance degradation bounded by a constant factor. They also compare performances of optimal solutions in both settings and show a lower bound of $\frac{5}{4}$. For more details on these results we refer to Section 4.4.

From a different perspective, there exist a lot of works dealing with algorithmic studies at the user level aiming at optimizing the execution times of applications (see for example [14, 42]). Our focus is at middle-ware level independent on the applications.

4.3 Problem Setting

Typical batch scheduling environments are composed of queues of jobs. The queue is defined as a set of jobs $\mathcal{J}$. Each job $j \in \mathcal{J}$ is characterized by a processing time p_j and a number $size_j$ of required resources (nodes or processors).

In this Chapter we decompose the platform into sets of closely located processors called *clusters* where each cluster is formed by a contiguous range of processors. The performance of communications is considered higher within a cluster than between two different clusters.

Different decompositions into clusters can be chosen for a given platform. The choice of this decomposition is left as a parameter in the batch scheduler. Setting this parameter within a target level will enable to decrease the amount of communications taking place at this level. We assume m processors partitioned into k different clusters.

In order to better illustrate this situation, we consider the example of a fat tree topology as displayed in Figure 4.1. Two different cluster configurations can be mapped on the tree hierarchy of Figure 4.1. We can choose to define clusters at the lowest hierarchical level to obtain four clusters $(Cl_1, \ldots, Cl_4)$ of size two. A second choice is possible by dividing one level above into two clusters (Cl'_1, Cl'_2) of size four.

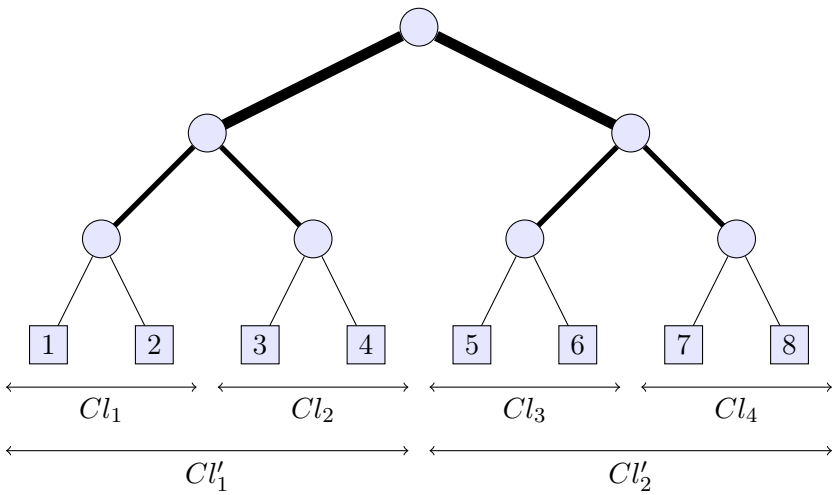


Figure 4.1 – Example of a fat tree interconnection

Based on the above definition of clusters we can highlight two key properties of scheduled jobs.

First, a job is *local* if it uses a minimal amount of clusters. For example, with respect to clusters $Cl_1, \ldots, Cl_4$ in Figure 4.1, a job j needing three processors is local when scheduled on processors $1, 2, 4$ and not local when scheduled across three different clusters. One of our main objectives is to achieve a high number of local jobs.

We also consider *contiguous* jobs. A job is contiguous when its allocated

processors form a contiguous range. Hence in the previous example, j is not contiguous because of the missing processor 3 in the range between 1 and 4. Contiguous jobs have the advantage to be always very close from being local using at most one additional cluster over the minimal amount required. Scheduling contiguous jobs does not require a fixed decomposition into clusters and provides therefore locality across all hierarchy levels simultaneously.

4.4 Worst-case Guarantees for Optimal Solutions

We provide in this section a worst-case analysis for comparing the optimal solutions with and without topological constraints (contiguity/locality). It is well known that FCFS with Backfilling guarantees no constant approximation ratio for makespan. In practice however it provides good performances with often a platform usage close to the optimal one. We intend here to study the impact of additional scheduling constraints like *contiguity* or/and *locality* on makespan in such configurations. More specifically, we are going to bound the worst-case ratio of the makespan of an optimal schedule when contiguity and/or locality are enforced over the makespan of an optimal schedule for the basic model where no additional constraints are demanded. As we will see, these ratios are within constant factors, providing in a way a good potential for the experimental results of Section 4.6.

In order to formalize the above comparison, let $C_{\max}$ be the value of the makespan of an optimal schedule for the basic model. Moreover, we denote by $C^C_{\max}$ and $C^L_{\max}$ the value of the makespan of an optimal schedule when only contiguity and only locality constraints, respectively, are enforced. Finally, let $C^{C+L}_{\max}$ be the value of the makespan of an optimal schedule when both contiguity and locality constraints are taken into account. We are interested in the ratios of the form

$$\max\{\frac{C^x_{\max}}{C^y_{\max}}\}$$

where $C^x_{\max}$ and $C^y_{\max}$ coincide with one of the quantities defined above. The maximum is taken over all instances and hence the ratio corresponds to the worst case.

4.4.1 Contiguous vs. Non-contiguous

In the case where locality constraints are not taken into account, then the relation between contiguous and non-contiguous schedules has been studied

in [15] where the following theorem has been proved.

Theorem 1. (Bładek et al. [15]) $\frac{5}{4} \leq \max\{\frac{C^C_{\max}}{C_{\max}}\} \leq 2$.

The authors in [15] provide both an upper and a lower bound which does not match each other, while they conjecture that the worst-case ratio is 5/4.

It has to be noticed also here that the same bounds hold when locality constraints are enforced for both contiguous and non-contiguous cases, since we can apply the same analysis for each cluster separately. Therefore, we have the following corollary.

Corollary 1. $\frac{5}{4} \leq \max\{\frac{C^{C+L}_{\max}}{C^L_{\max}}\} \leq 2$.

4.4.2 Local vs. Non-local

In this section we explore the relation between the optimal local and non-local schedules. In the results that will be presented, we consider that all clusters have the same size ℓ and all jobs can fit into a cluster, i.e., $size_j \leq \ell$ for each job $j \in \mathcal{J}$.

In the case where contiguity constraints are not taken into account, then the problem with locality constraints has been studied in [56] in the context of Resource Constrained Scheduling (RCS). The input of the RCS consists of a set of jobs and a set of processors. Each job j is characterized not only by its processing time $p_j \in \mathbb{Z}^+$ but also by a weight $w_j \in \mathbb{Z}^+$. Moreover, each processor i has a capacity $c_i \in \mathbb{Z}^+$ and it can execute many jobs in parallel under the constraint that at each time t the sum of the weights of the jobs executed by i at t does not exceed c_i.

RCS coincides with our problem when locality constraints are enforced. Indeed, each processor of the RCS problem corresponds to a cluster, while the weight of a job coincides with the number of processors that it requires, i.e., $w_j = size_j$. Using this relation we get the following theorem.

Theorem 2. $\max\{\frac{C^L_{\max}}{C_{\max}}\} \leq 2$. *This ratio is tight.*

Proof. Remy [56] has presented a 2-approximation algorithm for RCS. In fact, a stronger result has been proved: the makespan C of the created schedule for RCS is not more than twice the total load over the total capacity, i.e., $C \leq 2\frac{\sum_j p_j \cdot w_j}{\sum_i c_i}$. Due to the equivalence between the problems, this schedule corresponds also to a feasible schedule for our problem subject to

local constraints with makespan C. Clearly, $C \geq C^L_{\max}$. Moreover, the quantity $\frac{\sum_j p_j \cdot w_j}{\sum_i c_i}$ is a lower bound for $C_{\max}$, i.e., $C_{\max} \geq \frac{\sum_j p_j \cdot w_j}{\sum_i c_i}$. In total, we have that $\frac{C^L_{\max}}{C_{\max}} \leq 2$.

In order to prove the tightness, let us consider the following instance. We are given two clusters each one containing three processors, and three unit.

processing time jobs each one requiring two processors. The optimal schedule without any additional constraint is shown in Figure 4.2a and it has a makespan $C_{\max} = 1$, while the optimal schedule when locality constraints are enforced is shown in Figure 4.2b and it has a makespan $C^L_{\max} = 2$, and the theorem follows. □

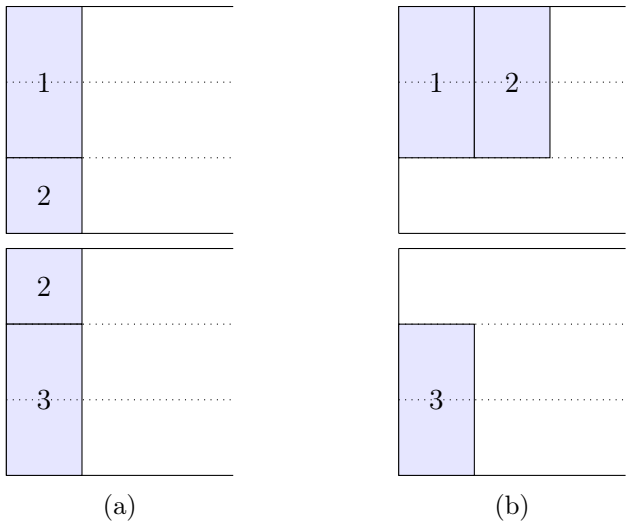


Figure 4.2 – Tightness example

We next study the relation between the optimal local and non-local schedules when contiguity is a required property for both. More specifically, we present a transformation from an optimal schedule without locality constraints to a schedule with locality constraints, and we prove the next theorem.

Theorem 3. $\max\{\frac{C^{C+L}_{\max}}{C^C_{\max}}\} \leq 2$. *This ratio is tight.*

Proof. For the upper bound, consider an optimal schedule $\mathcal{S}$ with makespan $C^C_{\max}$ in which contiguity constraints are enforced while locality constraints are not. We will partition the set of jobs $\mathcal{J}$ into two subsets: $\mathcal{J}_1$ contains

the jobs that are scheduled entirely into only one cluster, and $\mathcal{J}_2$ consists of the jobs whose execution takes place into two adjacent clusters (recall that we assume that all jobs require less processors than the size of a cluster).

We create the schedule $\mathcal{S}'$ with makespan $2C^C_{\max}$ as follows. In the interval $[0, C^C_{\max})$ we schedule the jobs in $\mathcal{J}_1$ exactly as they are scheduled in $\mathcal{S}$. In the interval $[C^C_{\max}, 2C^C_{\max})$ we schedule the jobs in $\mathcal{J}_2$. More specifically, consider a job $j \in \mathcal{J}_2$ which is executed in the interval $[t_1, t_2)$ using a set of $size_j$ processors from the clusters i and $i+1$. In $\mathcal{S}'$, we schedule j in the interval $[C^C_{\max} + t_1, C^C_{\max} + t_2)$ using an arbitrary set of $size_j$ processors only from the cluster i.

In order to show that the created schedule $\mathcal{S}'$ is feasible, observe that at each time t in $\mathcal{S}$ for each pair of consecutive clusters i and $i+1$, $1 \leq i \leq k-1$, there is at most one job that is executed at t using processors from both i and $i+1$, since the jobs in $\mathcal{S}$ satisfy contiguity. As we have assumed that for each job $j \in \mathcal{J}$ the number of required processors $size_j$ is smaller than the size of the cluster, in the schedule $\mathcal{S}'$ each job is executed in a set of contiguous processors of exactly one cluster. Finally, the optimal schedule with both contiguity and locality constraints has a makespan $C^{C+L}_{\max}$ at most the makespan of $\mathcal{S}'$, and hence the upper bound follows.

For the tightness, we can use the same example as in the proof of Theorem 2. □

4.4.3 Topological-aware vs. Basic Model

In this section we study the relation between the optimal solutions of the basic model without constraints and the case where contiguity and locality are both required. By combining Corollary 1 and Theorem 2 or Theorems 3 and 1, we directly get that $\max\{\frac{C^{C+L}_{\max}}{C_{\max}}\} \leq 4$. In what follows, we improve the above bound. As in the previous section, we consider that all clusters have the same size $\ell = \frac{m}{k}$ and all jobs can fit into a cluster, i.e., $size_j \leq \ell$ for each job $j \in \mathcal{J}$.

Our approach here is to present an algorithm with bounded worst case approximation ratio for the problem with contiguity and locality constraints. In the computation of this approximation ratio, we use as a lower bound for the value of the optimal solution the total processing requirement $A = \sum_{j \in \mathcal{J}} p_j \cdot size_j$ divided by the total number of available processors m, as well as, the maximum processing time $p_{\max} = \max\{p_j : j \in \mathcal{J}\}$ among all jobs. The crucial observation here is that both these quantities are lower bounds also for the value of the optimal solution for the basic model. Hence, we can obtain the desired result. In order to create the approximation al-

gorithm, we combine two basic ingredients: an approximation algorithm for the Strip Packing (SP) problem [19] and an approximation algorithm for the classical $P||C_{\max}$ problem of scheduling serial tasks [29]. Several algorithms are known for both SP and $P||C_{\max}$. However, the choice of the algorithms which we will use is driven by the lower bounds used by them, since at the end we need to compare with $\frac{A}{m}$ and $p_{\max}$.

The input of the SP problem consists of a unit-width, infinite-height strip and a set of rectangles $\mathcal{R}$ parallel to the bottom of the strip. Each rectangle $j \in \mathcal{R}$ is characterized by a width w_j, $0 < w_j \leq 1$, and a height $h_j > 0$. The goal is to pack the rectangles into the strip such that no two rectangles overlap and the height of the strip used is minimized. A well-known algorithm for the SP problem is the Next-Fit Decreasing-Height (NFDH) algorithm which considers the rectangles in non-increasing order with respect to their heights and applies the Next-Fit policy using this order. More specifically, the algorithm keeps at each time during its execution a set of batches. Let B_i be the i-th batch that NFDH opens and $W_i = \sum_{j \in B_i} w_j$ and $H_i = \max_{j \in B_i}\{h_j\}$ be the width and the height, respectively, of B_i. According to the Next-Fit policy, if the currently handled rectangle z fits to the latest opened batch B_i, i.e., $W_i + w_z \leq 1$, then we add z to B_i; otherwise we open a new batch B_{i+1} and we add z to it. The following lemma has been proved in [19].

Lemma 1. (Coffman et al. [19]) *NFDH packs a set of rectangles in a strip of total height* $H = \sum_{i=1}^{b} H_i \leq 2 \sum_{j \in \mathcal{R}} w_j \cdot h_j + H1$, *where* b *is the number of created batches.*

The input of $P||C_{\max}$ consists of a set of m' parallel processors and a set $\mathcal{J}'$ of jobs each one characterized by a processing time p'_j. The goal is to find a schedule of minimum makespan. Let $p'_{\max} = \max\{p'_j : j \in \mathcal{J}'\}$ be the maximum processing time among all jobs in $\mathcal{J}'$. A classical algorithm for $P||C_{\max}$ is the well known List Scheduling (LS) algorithm which at each time where a processor becomes idle, it selects a job which is not yet scheduled and it assigns it to the idle processor. The following lemma has been proved in [29].

Lemma 2. (Graham [29]) *LS returns a schedule of makespan* $C \leq \frac{1}{m'} \sum_{j \in \mathcal{J}'} p'_j + (1 - \frac{1}{m'}) p'_{\max}$.

The idea of our algorithm is to initially transform an instance of our problem to an instance of SP and apply NFDH. More specifically, for each job $j \in \mathcal{J}$ we create a rectangle $j \in \mathcal{R}$ with $w_j = \frac{size_j}{\ell}$ and $h_j = p_j$. The

result of the application of NFDH to the created instance is a set of batches $B_1, B_2, \ldots, B_b$, that are sorted in non-increasing order with respect to their heights, i.e., $H_1 \geq H_2 \geq \ldots \geq H_b$, and their total height satisfy Lemma 1. Using these batches, we create an instance of $P||C_{\max}$ as follows: for each batch B_i we create a job $i \in \mathcal{J}'$ with processing time $p'_i = H_i$. Moreover, we consider $m' = k$ processors. Finally, we apply the LS algorithm and we transform the created schedule to a feasible schedule for our initial instance. In fact, each processor of $P||C_{\max}$ corresponds to a cluster and each job in $\mathcal{J}'$ corresponds to a set of jobs in $\mathcal{J}$ that can be feasibly executed in parallel on a single cluster.

Theorem 4. *There is an algorithm for the problem with contiguity and locality constraints which returns a schedule with makespan* $C \leq 3\max\{\frac{A}{m}, p_{\max}\}$.

Proof. By using Lemma 2, for the makespan C of the schedule created by our algorithm it holds that:

$$\begin{aligned} C &\leq \frac{1}{m'}\sum_{j\in\mathcal{J}'} p'_j + (1-\frac{1}{m'})p'_{\max} \\ &= \frac{1}{k}\sum_{i=1}^{b} H_i + (1-\frac{1}{k})H_1 \end{aligned}$$

Hence, by using Lemma 1 we get:

$$\begin{aligned} C &\leq \frac{1}{k}(2\sum_{j\in\mathcal{R}} w_j \cdot h_j + H_1) + (1-\frac{1}{k})H_1 \\ &= \frac{1}{k}(2\sum_{j\in\mathcal{J}} \frac{size_j}{\ell}\cdot p_j + p_{\max}) + (1-\frac{1}{k})p_{\max} \\ &= 2\frac{A}{m} + \frac{p_{\max}}{k} + (1-\frac{1}{k})p_{\max} \\ &\leq 3\max\{\frac{A}{m}, p_{\max}\} \end{aligned}$$

□

Theorem 5. $2 \leq \max\{\frac{C_{\max}^{C+L}}{C_{\max}}\} \leq 3$.

Proof. The lower bound follows from Theorem 2.

For the upper bound, we observe that the makespan of the optimal solution with contiguity and locality constraints is at most equal to the makespan of the approximation algorithm proposed above. Hence, by Theorem 4 we have that $C_{\max}^{C+L} \leq C \leq 3\max\{\frac{A}{m}, p_{\max}\}$. As $C_{\max} \geq \max\{\frac{A}{m}, p_{\max}\}$, the theorem follows. □

4.5 Allocation Algorithms

In this section we present new variants of the classical Backfilling algorithm designed to achieve improved job locality. Different algorithms allow the administrator to choose more or less strict contiguity and locality constraints. All proposed algorithms are designed to be easily integrated in existing systems.

4.5.1 Backfilling

Most of the scheduling schemes currently in use are based on variable partitioning [25]. In this scheme, each job receives a partition of the machine containing a subset of processors. One particular way partitions can be allocated is First Come, First Serve (FCFS). One notable disadvantage of using FCFS to allocate partitions is fragmentation, where free processors cannot be used for the following jobs and remain idle until more processors are available. As a result, systems that employ FCFS tend to have low system utilization rates [31, 35].

Another approach is to use non-FCFS policies, reordering jobs in the queue as they are submitted. In this scenario, the scheduler improve utilization by allowing jobs to move forward in the queue. However, this procedure, if overly utilized, can lead to starvation of big jobs. The common solution to this problem is to allow only a certain number of jobs to move forward in the queue before the next big job is allocated.

Figure 4.3 shows an example of FCFS allocation. The horizontal axis shows the schedule time, denoted by t. The Figure shows that some jobs are executing and the next job in the queue needs all the processors available in the machine. Since no jobs are allowed to move forward, processors will become idle as they are freed by the jobs that are currently running. In the worst case, the total number of available processors in the machine minus one will be idle, moments before the big job starts.

A more complex approach is to require users to submit an estimation of the run time of their jobs. The scheduler, in turn, can consider this information to allow short jobs to move forward in the queue and fill gaps left by the

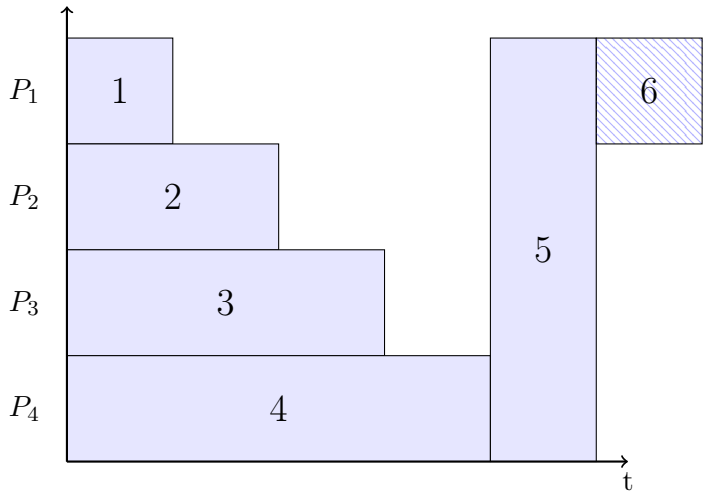


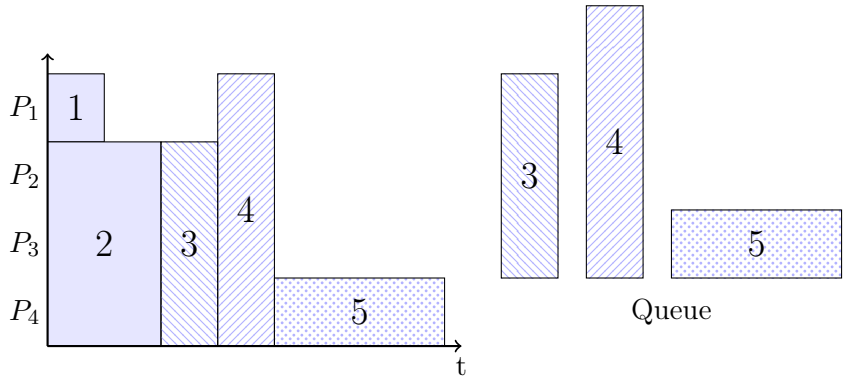
Figure 4.3 – Example of FCFS allocation scheme

allocation of big jobs. This scheduling scheme is called Backfilling, and was developed for the IBM SP1 parallel supercomputer installed at the Argonne National Laboratory as part of EASY (Extensible Argonne Scheduling sYstem) [43, 51]. Users are expected to provide estimated run times that are as accurate as possible, since a low estimation can lead to the job being killed and canceled before it terminates, while a high estimation may lead to long waiting times for the other jobs and additional fragmentation.

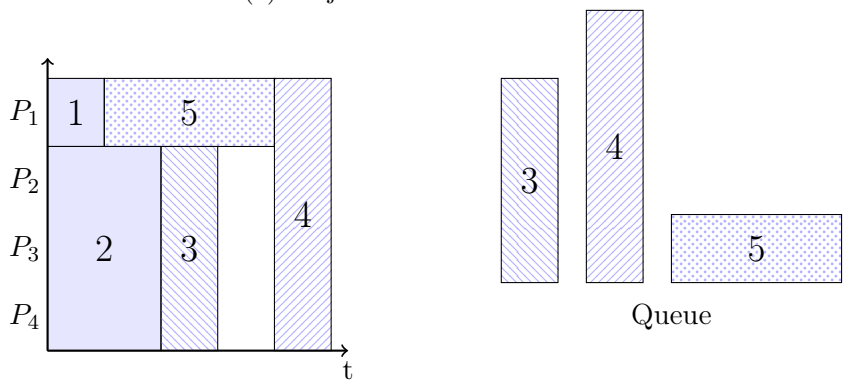
The EASY Backfilling algorithm checks only one thing: that jobs that move forward in the queue do not delay the first queued job. This approach is shown to lead to unbounded queuing delays for other queued jobs. Therefore, it prevents the system from making predictions regarding when each job—with the exception of the first job of the queue—will run.

An example can be seen on Figure 4.4. The first part of the Figure shows the allocation if no jobs are allowed to move forward in the queue. But, since the second job can be executed at $t = 2$ without delaying the first job in the queue, it is allowed to move forward. As a consequence, the third job is delayed.

An alternative to EASY Backfilling is to only move short jobs that do not delay any job already in the queue. It has been shown that, for the workloads measured on SP2 systems, EASY Backfilling provides better performance, with the penalty of not having estimated starting times for the jobs. In contrast, analyzing workloads for different systems it is possible to see that both algorithms have similar performances. In this case, Conser-



(a) No jobs are allowed to move forward



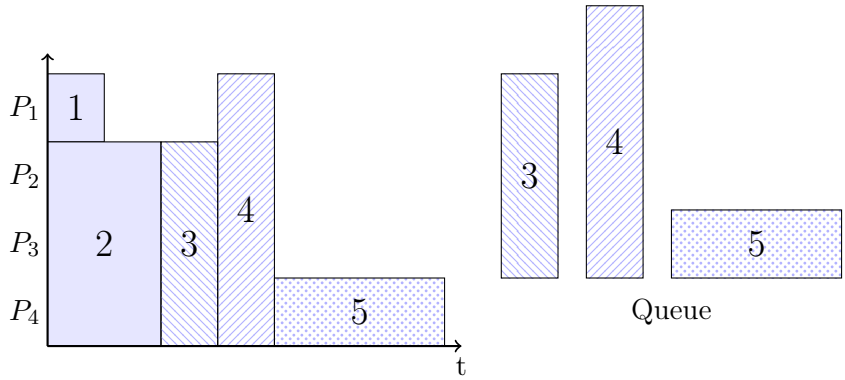
(b) Job 5 moves forward without delaying the first job in the queue

Figure 4.4 – Example of EASY job allocation

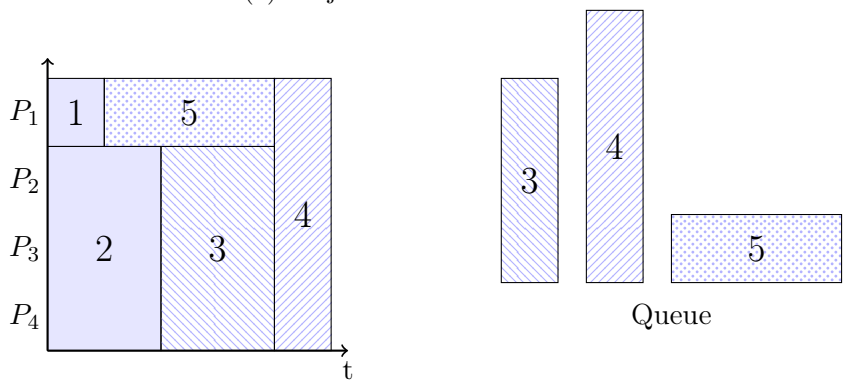
vative Backfilling is preferable to the EASY algorithm due to the improved predictability associated with knowing the starting times of the jobs.

Figure 4.5 shows an example of allocation using Conservative Backfilling. The first part of the Figure shows how the allocation would be if Conservative Backfilling is used with the same jobs as in Figure 4.4, that showed an example of allocation using EASY Backfilling. In this case, the third job is not allowed to move forward since there is no space for it without delaying the second job. The second part of the picture shows another scenario where there is enough space for the third job to be backfilled and start running sooner, without delaying any of the subsequent jobs.

Conservative Backfilling is the most commonly version mentioned in the literature [25, 27]. In this version, backfilling is done only if it does not delay *any* of the previous jobs in the queue. Users provide each submitted job



(a) No jobs are allowed to move forward



(b) Job 5 moves forward without delaying the first job in the queue

Figure 4.5 – Example of allocation using Conservative backfilling

with a running time limit and a fixed number of processors. This knowledge allows the scheduler to know in advance all resources consumption and to backfill jobs into inactivity periods. The main feature is that no starvation occurs in Conservative Backfilling. Moreover, all jobs receive a reservation period when they are submitted. This means that the job is guaranteed to start executing either at that time or before, but never later than the reservation.

Figure 4.6 displays a schedule example with the corresponding division into time slices. For each time slice T_i, let $S[i]$ be the set of the current idle processors during T_i. Moreover, we denote by $|S[i]|$ the cardinality of $S[i]$.

When scheduling a job, the Backfilling algorithm starts by finding the first contiguous set of time slices large enough for both the number of required processors and run time. Simple pseudocode for this operation is given in

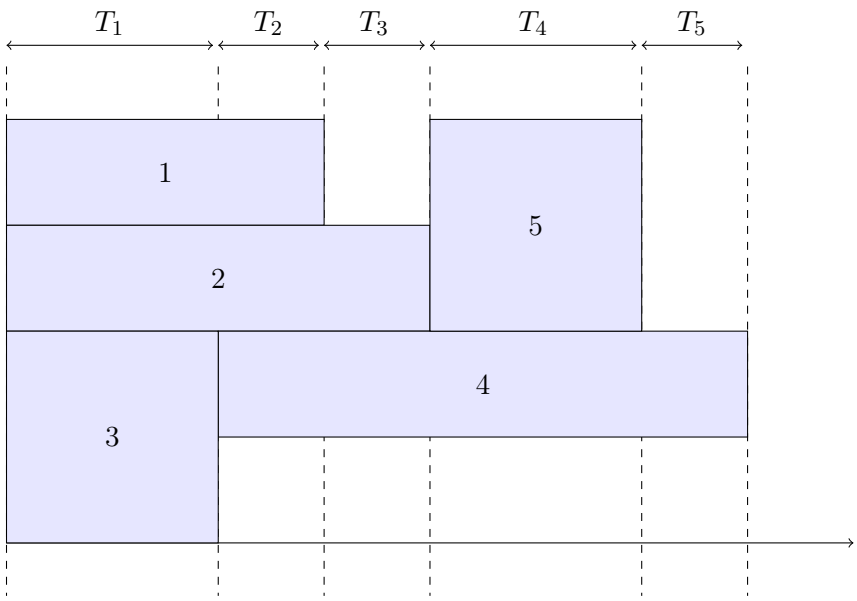


Figure 4.6 – Schedule representation through time slices

Algorithm 1.

4.5.2 Algorithms

One of the final steps of the Backfilling algorithm (as shown in Algorithm 1, line 15) is to reduce the set P of available processors to the required number of processors. This operation is called the *allocation* step.

All of our algorithms are based on the specialization of this step in order to select specific processors. It is even possible for the allocation to return an empty set of processors (when no set matches required properties). In this case the search for valid time slices will continue by resuming the iterative process in the main algorithm.

We present below the following algorithms:

- basic,
- best effort contiguous,
- forced contiguous,
- best effort local,

Algorithm 1: Backfilling

Data: job j, processing time p_j, required processors number $size_j$, list of time slices S

Result: list P of available processors for the job, starting time t

```
foreach time slice S[i] do
    if |S[i]| < size_j then
        next;
    end
    P ← S[i];
    time ← 0;
    foreach time slice index j starting from i do
        P = P ∩ S[j];
        time ← time + duration(S[j]);
        if time ≥ p_j then
            break;
        end
    end
    if |P| ≥ size_j then
        P ← size_j processors of P;
        t ← starting_time(S[i]);
        return (P, t)
    end
end
```

- forced local.

Basic Backfilling (Algorithm 2) This is the simplest form of the allocation algorithm. It chooses the first processors that are available, whether they are contiguous or not. This policy is commonly used in most batch-schedulers.

Backfilling with best effort contiguity (Algorithm 3) This algorithm aims to reduce the processors available in the execution profile by finding a block of contiguous processors. If there are no possible contiguous blocks, a non contiguous block is assigned.

Algorithm 2: Basic backfilling allocation algorithm

```
Data: job size size_j, set P of available processors
Result: block of size_j processors
sort P by processor rank;
return P[1]..P[size_j]
```

Algorithm 3: Backfilling with best effort contiguity

```
Data: job size size_j, set P of available processors
Result: block of size_j processors
sort P by processor rank;
first ← 1;
length ← 1;
for i ← 2 to |P| do
    if P[first]..P[i] is a contiguous block then
        length ← length + 1;
    else
        first ← i;
        length ← 1;
    end
    if length = size_j then
        return P[first]..P[first + size_j]
    end
end
return P[1]..P[size_j]
```

Backfilling with best effort locality (Algorithm 4) This algorithm aims to reduce the number of clusters used to schedule the job to the minimal one. If there are not enough available processors in the clusters such that this constraint can be applied, the job is scheduled on the block of processors that are as close as possible to the one it aimed for earlier. This algorithm is shown in 4.

Backfilling with forced contiguity This algorithm reduces the list of available processors making sure that the reduced list is made of a contiguous block of processors. In the case that there are no possible contiguous blocks, the current starting time slice is rejected and a new posterior one has to be found. This algorithm is implemented similarly to the best effort contiguous

Algorithm 4: Backfilling with best effort locality

```
Data: job j, set P of available processors
C ← array of clusters (initially empty);
foreach processor i in S do
    let j be the index of the cluster containing i;
    assign i to C[j];
end
sort C by the number of processors in each cluster in decreasing order;
return size_j processors of the first k clusters in C verifying
  Σ_{i=1}^{k} size(C(i)) ≥ size_j
```

one, with the exception that when a contiguous block of processors cannot be found, an empty block is returned so that the starting time slice is rejected.

Backfilling with forced locality This algorithm reduces the list of available processors making sure that the new list comprises a block of processors that uses the minimum possible number of clusters. Similarly, if such block cannot be found, an empty block is returned so that the starting time slice is rejected. It is implemented like the best effort local algorithm, with the difference that if there are less than $size_j$ processors in the first k clusters, an empty block is returned instead of the first $size_j$ processors.

It should be noticed that in the standard Backfilling algorithm the way to achieve the allocation is not fully specified. Thus one scheduling instance might yield several different solutions. In particular the classical algorithm is able to give exactly the same solution as *basic*, *best effort contiguous* and *best effort local*. This property gives a strong argument in favor of a switch from actual implementations towards one of our proposed allocation algorithms.

4.6 Experimental Results

In this section we compare the performances of the different algorithms proposed under realistic scenarios. All algorithms have been implemented in a custom discrete events simulator. This simulator works by reading a batch scheduler trace file in the SWF format [9] and extracting for each job the number of required processors and the execution time. Trace data are extracted from three different trace archives: CEA-Curie [11], METACENTRUM [6] and RICC [7].

In our experiments we consider only *real* execution times. This choice is common when doing simulations since user submitted times are strongly overestimated. For instance, one third of the jobs in the Curie trace have user submitted times of 24 hours (default setting).

Our experiments are designed by setting:

- the machine parameters: number of processors and cluster size,
- the number of jobs,
- the number of experiments instances,
- the algorithms to compare.

We start by choosing the platform. We tested varying numbers of processors from 64 to 1000 and we present in this Chapter the results for 512 processors which correspond to the worst performances for our algorithms. For this virtual platform we tested results for different cluster sizes from 16 to 128. The results are presented for a cluster size of 16, while similar results are achieved for larger sizes.

Jobs selection is handled in the following way: we first filter the trace by removing all jobs requiring more processors than the machine size. Then we randomly select a fixed number of jobs from the remainder. A wide range of values for the number of jobs is tested and we present here results for 300 jobs. Instances for higher number of jobs lead to similar results. We extract 512 job instances from the trace and we use these instances as input for each experiment.

The following algorithms described in Section 4.5 are compared: *basic*, *best effort contiguous*, *best effort local*, *forced contiguous* and *forced local*.

Our experiments are designed to quantify and make explicit the trade-off between locality and makespan. Performance degradations are expected on classical scheduling objectives as locality constraints grow. On the other hand, a gradual improvement of job locality should be observed.

Since the real amount of communication for each job is unknown we cannot estimate the actual gains/losses in processing time due to different job placements. We therefore look at makespan and locality as two separate objectives. We assume than improvements in locality will at the end translate into improvements in makespan. One common set of experiments is run for both objectives.

4.6.1 Makespan

We start by presenting results for the makespan experiments for 512 instances of 300 random jobs running on a 512 processors platform. The results presented here are generated with jobs extracted from the Curie trace. Experimenting with other traces gives very similar results which are therefore not presented here.

Initially, a comparison of the *basic* Backfilling algorithm with *best effort contiguous* is given. *Best effort contiguous* creates few changes over *basic* and it is a good candidate for starting our comparisons. These results are shown in Figure 4.7. Each point represents an instance with the corresponding makespan for *basic* on the x-axis and for *best effort contiguous* on the y-axis. Since most points are located on the $y = x$ line it is clear that both algorithms yield similar performances. In fact, their average performance differs by less than 0.1%.

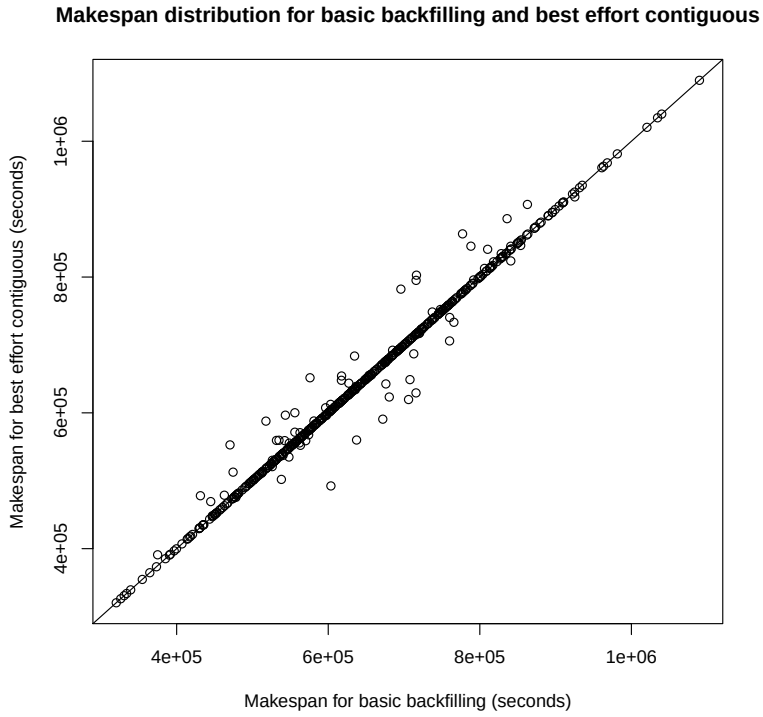


Figure 4.7 – Makespan comparison: basic Backfilling to best effort contiguous.

In Figure 4.8 and Figure 4.9 we compare the *basic* Backfilling algorithm with *forced contiguous* and *forced local*, respectively. These figures are similar to the previous one with the makespan of the *basic* algorithm on the x-axis and the makespan of *forced contiguous* and *forced local*, respectively, on the y-axis. *Forced local* shows a similar behavior as *best effort contiguous*. On the other hand, the shape of the cloud of points displayed for *forced contiguous* thickens. This particularity indicates that results are less stable for this algorithm. Note however that its performance is still close to the one of the *basic* algorithm with under two percents of difference in average.

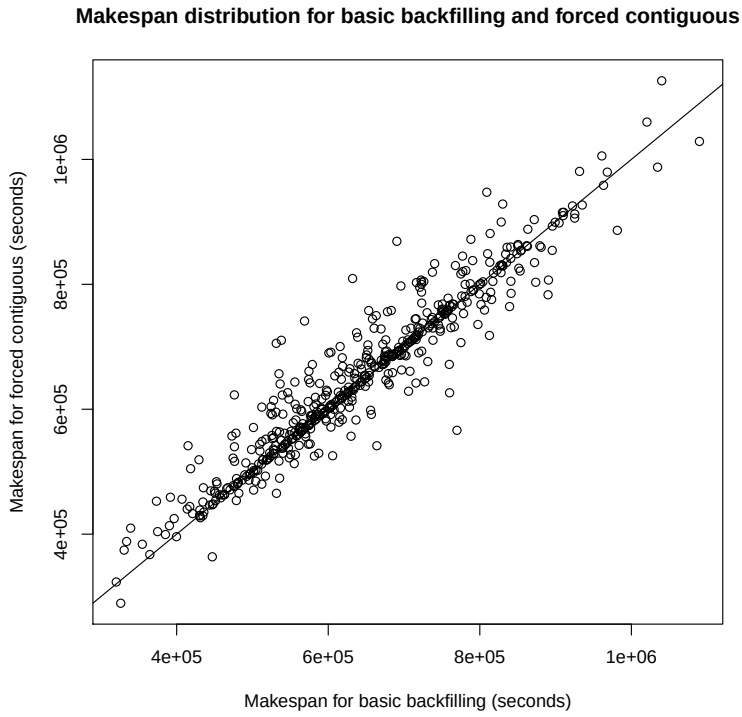


Figure 4.8 – Makespan comparison: basic Backfilling to forced contiguous.

The results for *best effort local* are not presented here but it compares to *basic* in a similar way as *forced local*.

As a conclusion, the makespan experiments show that all proposed algorithms yield similar performances as *basic*. This result is very positive and shows a good application potential since additional constraints generate negligible extra costs. However, the experiments indicate that enforcing

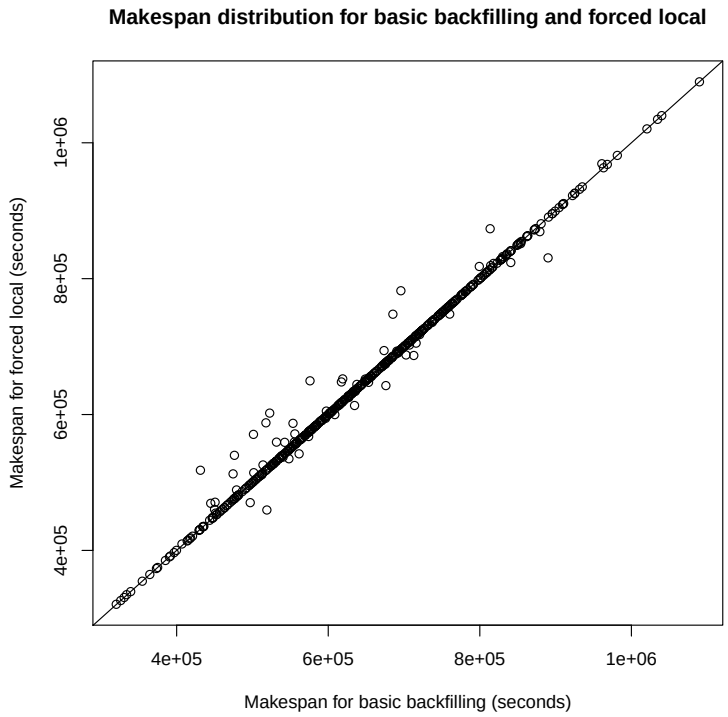


Figure 4.9 – Makespan comparison: best effort contiguous to forced local.

contiguity leads to less performance stability.

4.6.2 Contiguity and Locality

First we compare in Figure 4.10 the number of contiguous jobs for each algorithm (with the exception of *forced contiguous* which achieves by definition 100% of contiguous jobs). For each algorithm we plot the probability density function of the number of contiguous jobs over all instances. We observe that *basic* and *best effort local* have similar behaviors. *Forced local* improves the number of contiguous jobs, closely followed by *best effort contiguous*. This corresponds to the results expected from the design of the algorithms.

We continue by studying locality which is the main target of our work. In Figure 4.11 the probability density function of the number of local jobs is plotted. We can see here that *best effort contiguous* achieves worse results than *basic*. *Forced contiguous* outperforms *basic* with however more variabil-

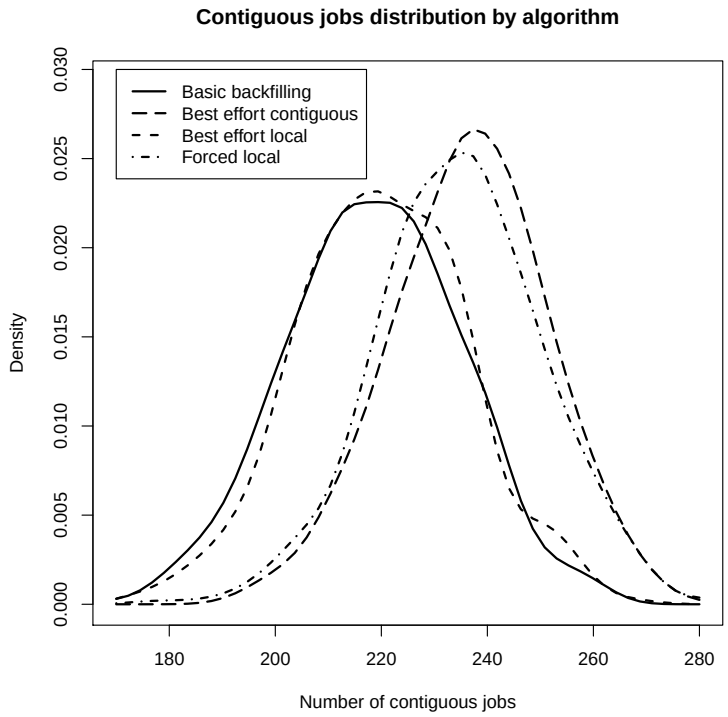


Figure 4.10 – Distribution of the number of contiguous jobs by algorithm.

ity in the performance. Finally *best effort local* achieves the best results by scheduling jobs locally 90% of the time.

The results for *best effort contiguous* are unexpected. They can be explained by the fact that contiguity guarantees us to use a number of clusters which is at most one more than the optimal one. However such a job is not considered as local.

In order to get a better view of the achieved locality of jobs we propose to consider a new metric, the *locality ratio*.

We consider a random variable $K_{A,I}$ which is the number of clusters in use for a random job in a schedule of instance I by algorithm A. Similarly we consider a random variable ${K_I}^*$, the minimal number of clusters required for a random job taken uniformly in instance I. The *locality ratio* for an algorithm A and an instance I is defined as the ratio of expectations: $\frac{E(K_{A,I})}{E({K_I}^*)}$.

Figure 4.12 displays the density function of locality ratios for all algorithms. *Forced local* is not displayed in this figure since all its jobs are

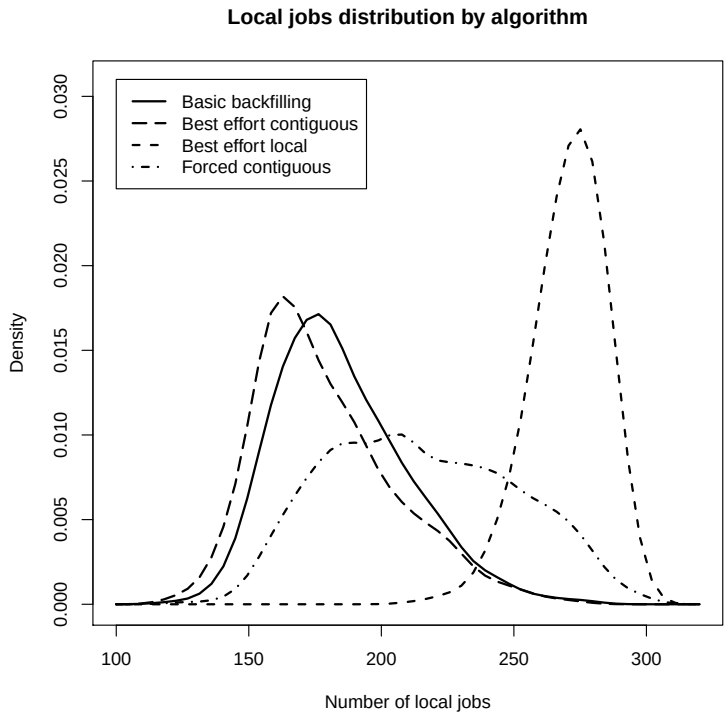


Figure 4.11 – Distribution of the number of local jobs by algorithm.

local. Figure 4.12 shows that with respect to the locality ratio all our algorithms proceed as expected. *Best effort local* gives the best results, followed by *forced contiguous*. *Basic* and *best effort contiguous* show similar performances with a slight advantage for *best effort contiguous*.

Similar results are achieved for varying cluster sizes and trace files with similar performances for the different algorithms. As such they are not presented within this Chapter.

4.7 Conclusion

In this Chapter we take interest into scheduling jobs with *contiguity* and *locality*.

We provide a theoretical analysis of the impact of these constraints on makespan. By using bounds on optimal makespan ratios we characterize the worst case impact of different sets of constraints on "good" solutions.

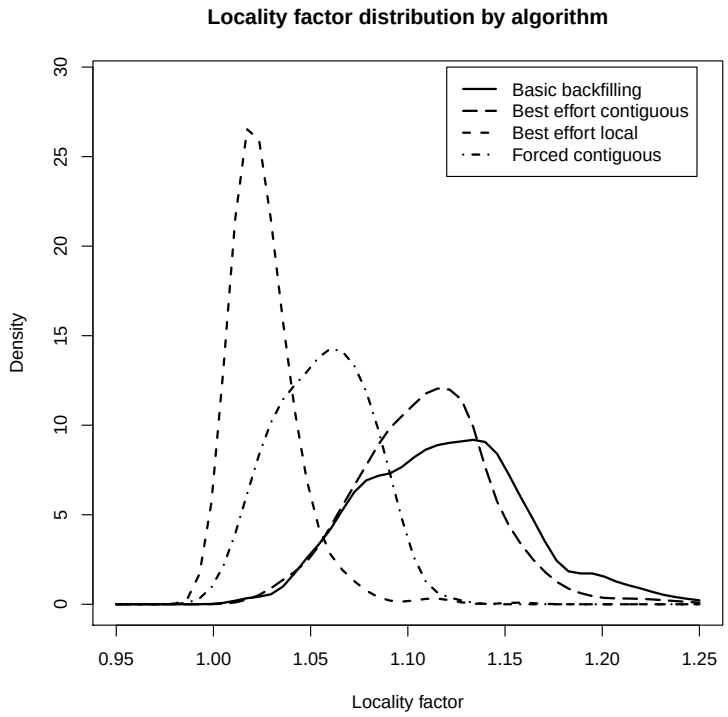


Figure 4.12 – Distribution of the locality ratio by algorithm.

We show for all cases that this impact is limited to a constant ratio. We present a compilation of existing results on contiguity and locality completed by new results. More specifically we provide a 3-approximation algorithm for scheduling contiguous and local jobs and some tightness results for the other cases.

Moreover, two different ways to adapt the allocation step of the standard FCFS with Backfilling algorithm to contiguity and locality are presented.i More specifically, we distinguish between best effort algorithms and a more strict enforcing of constraints.

Our simulation campaign shows that the proposed algorithms do not affect the makespan in a negative way. All algorithms with the exception of *forced contiguous* show a strong stability with more than 99.6% of instances with a makespan deviation of less than 2% from the *basic* one. We are also able to improve locality with especially strong improvement for *forced contiguous*, *best effort local* and *forced local*. Experiments show that very

simple constraints can indeed achieve a more local job allocation with little or even no information on platform topology as in the case of *forced contiguous*.

We consider our results to be very promising and envision different ways of further improvements. One of the first possible research direction is to implement our work in a real batch scheduler like SLURM.

We also wish to be able to characterize communications within jobs in order to evaluate the actual performance or energy gains which can be achieved by our algorithms. This would require some platform-wide monitoring of communications over large periods of time in order to extract realistic models of communications within jobs for a wide set of applications.

Chapter 5

Improving Backfilling with Full Locality Awareness

5.1 Introduction

In this Chapter we are interested in studying an improvement to the Contiguity and Basic Locality constraints studied before. The Backfilling algorithm, combined with the FCFS policy, works very well when there are no latency issues between the processors allocated to a job. However, performance decreases considerably when the latency between some of the processors is high enough to impact the jobs' run times. The reason for this is that the Backfilling algorithm does not choose specific processors for each job. Depending on the implementation and the data structures being used, it selects the processors for a job in different ways. Some examples are picking the first processors from a list ordered by increasing ids or taking them randomly from a hash.

Therefore, scheduling jobs with Backfilling can be significantly improved by adding to it constraints that help better choose which processors each job should be assigned to.

As far as we know, most of the approaches available in the literature that present some form of topology awareness do it in one of two ways. The first is to add to a job a library that can detect the different elements of the platform, such as topology and performance of the processors. Then, the idea is to use this information to reorganize the job and match the computing power and communication costs to the characteristics of the job. The limitation of this approach is that it has no power over the initial assignment, which may have been inadequate. The second most common approach is

to include in the job submissions information about communication in the form of communication matrices. This approach allows the scheduler to place close together processes that have the most communication between each other. The problem, in this case, is that the communication matrix has limited knowledge of the communication done between processes. It contains information either about the frequency of communication or message size. There is no way to differentiate between processes that communicate in short burst and at constant rates. Furthermore, this approach requires jobs to be profiled before they are submitted. Often, small changes in the code or in the number of processes require that a new profile is made, which is commonly done by executing the job beforehand.

Apart from the Contiguity and Basic Locality constraints that were studied in the previous Chapter, Full Locality is a constraint that considers the topology of the platform, and not just the fragmentation of assignments or the size of each group of processors at the last level. The main goal is then to utilize this information to assign jobs to regions of the platform where communication costs between the processors assigned to a particular job are minimized. This constraint does not require jobs to be assigned contiguously, since communication costs between processors that are located at the last level are considered to be the same, for all processors in the group.

In order to compare the performance of the Full Locality constraint to the previously proposed ones, the discrete events simulator was updated. The new constraint was added and the simulator was modified to utilize original traces with as few modifications as possible. To achieve this, the simulator now works in an online fashion, where knowledge of the jobs' real run time is limited to the part that handles the events, and instead bases the scheduling decisions on wall times supplied by users. These wall times, as we know, can be deeply flawed, either by overestimation or underestimation, which can lead to the job being killed before it finishes its execution.

To deal with the problem related to imprecise wall times given by the users, we implement in the simulator the feature of reassignment of jobs [1]. When a job finishes early, either due to overestimation of the wall time or an improvement of the run time due to better locality, the simulator releases the processors for that job and checks, for each job with a reservation that has not started yet if that job can start now. If it can, it is moved forwarded, otherwise it keeps its reservation.

Furthermore, we prepare a campaign of experiments, where different platforms and their corresponding traces are utilized to compare Full Locality

[1] also called schedule compression

to the other constraints. To that end, the original traces are split in smaller periods that are in turn used as input in the experiments. The generated traces and other parameters are combined to run the experiment campaign.

Experiment results indicate that the Full Locality constraint is an improvement over the previously proposed ones and specially Basic Backfilling, when considering several metrics, for each one of the considered platforms.

This Chapter is organized as follows. Section 5.2 discusses the relevant related works present in the literature concerning locality awareness. The problem description are presented in Section 5.3. Sections 5.4 and 5.5 describe some of the modifications done to the simulator for this work. Then, Section 5.6 presents the Full Locality algorithm. Finally, Section 5.7 presents the experiment campaign as well as the results when comparing the Full Locality constraint to the previously proposed ones and Basic Backfilling and a brief conclusion is given in Section 5.8.

5.2 Related Work

Algorithms based on Backfilling are the most commonly used scheduling algorithms in batch schedulers [32, 61]. All variations of the Backfilling algorithm work by assigning jobs in a queue by incoming order to computing resources.

The TreeMatch core algorithm takes as input a matrix modeling the amount of communication between the processes of the job. It also needs a representation of the underlying architecture modeled as a tree. As with our model, the leaves contain processors where the job can be executed.

The communication matrix represents the target application's communication pattern. It consists of the global amount of data exchanged between each pair of MPI processes and is stores in a $p \times p$ communication matrix. Table 5.1 shows an example of communication matrix of a MPI application where each processors communicates with its neighbors.

In order to gather this data, the team chooses to introduce a slight amount of profiling code within an existing MPI implementation. By doing this, they are able to trace data exchanges both in point-to-point and collective communication. The main drawback to this approach is that a preliminary run of the application is mandatory in order to get the communication data. Additionally, any change in the execution (number of processors, input data, etc.) generally requires a new execution of the application to generate a new trace.

The goal of TreeMatch is to assign each MPI process to a processor. In

P	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	100	0	0	0	0	0	100
1	100	0	100	0	0	0	0	0
2	0	100	0	100	0	0	0	0
3	0	0	100	0	100	0	0	0
4	0	0	0	100	0	100	0	0
5	0	0	0	0	100	0	100	0
6	0	0	0	0	0	100	0	100
7	100	0	0	0	0	0	100	0

Table 5.1 – Example of communication matrix where each processors communicates with its neighbors

other words, TreeMatch attempts to match the available processors to the leafs of the tree on the platform. In order to improve the communication time of a MPI application, the TreeMatch algorithm maps processes to processors depending on the amount of data they exchange during execution.

TreeMatch is a recursive algorithm. It processes the tree upward starting at the lowest level. At this depth the arity of the node is two. The algorithm then generates all the combinations of two processes. For example, process 0 can be combined with processes 1 to 7 and process 1 with processes 2 to 7 and so on. The objective is to find four groups of two processes that do not have processes in common. To find these groups, a graph of incompatibilities is built. Two groups are incompatible if they share a process. In this graph, vertices are groups and an edge exists between two vertices if the corresponding groups are incompatible. The desired set of groups is then an *independent set* of this graph.

Additionally, not all combinations of groups of processes are of equal quality for the purposes of the algorithm. The quality of the combination depends on the values of the matrix. To account for this, each vertex receives the value of the combined communication between the group and the other processes. The smaller this value, the better the combination. Since finding such an independent set of minimum weight is NP-Hard and in-approximable at a constant ratio [36], heuristics are used to find “good" independent sets.

5.3 Problem Modeling

Typical batch scheduling systems are composed of queues of jobs. Each queue is defined as a set of jobs J. Each job j in the queue is characterized

by a run time p_j, a wall time w_j that is submitted by the user and a number $size_j$ of required processors.

5.3.1 Platform

In this work we model the platform as sets of closely related processors connected to a network unit and call each set a *cluster*. In other words, a cluster is a contiguous range of processors. Clusters, in turn, are connected to each other by network units.

Each platform has its own composition of network units and processors. This composition depends on how the platform was constructed and how the nodes are connected.

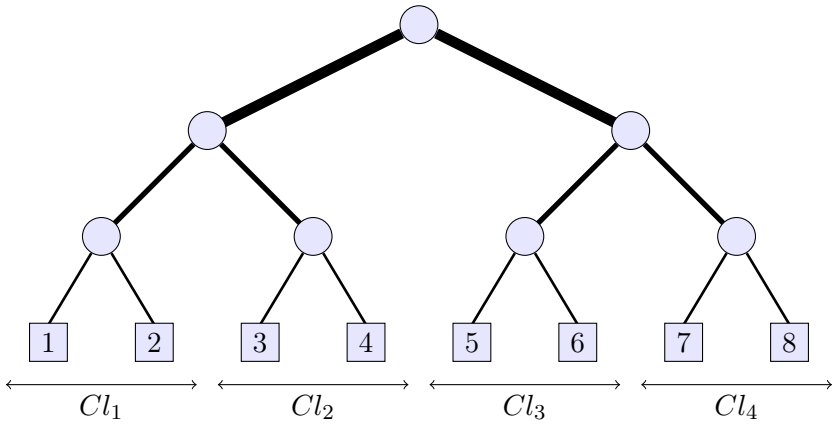


Figure 5.1 – Example of a tree-shaped topology

To help illustrate, we consider the example of a tree topology shown in Figure 5.1. The circles are network units — normally in the form of switches — and the squares are processors. The clusters are located at the lowest level of the tree, denoted Cl_1 to Cl_4. If all the processors allocated to a job that needs two processors are in the same cluster, for example Cl_1, the communication cost is considered to be the lowest possible in the platform. If the processors allocated to the job are distributed between two clusters, for example Cl_1 and Cl_2, the communication between the processors needs to pass through one level above the minimum for a job with two processors. Since this level has a higher communication cost than the previous one, the job will have a longer running time. Finally, if the job has processors allocated on both sides of the tree, for example in clusters C_1 and Cl_3, the communication has to go through the root, which is the path with the highest

cost. In this case the job's run time will be the longest.

Based on this definition of cluster, we can describe some key properties of scheduled jobs.

5.3.2 Backfilling

In the previous Chapter, on Section 4.5.1, we discuss the characteristics of the Backfilling algorithm, its goals, advantages and limitations. In this Chapter, we employ the same basic algorithm in order to serve as a starting point for our new proposed improvements and variants. The differences in the implementation as well as the new additions are detailed in the following Sections.

5.3.3 Success Rate

We consider a job's success rate its status after finishing execution. This metric is important since it impacts directly the user that submitted the job. When the user submits the job he/she expects a certain run time and decides on a wall time accordingly. If the job is placed in a platform level that is worst than the minimum, the run time will increase and might reach the wall time. When that happens, the job fails to finish and is canceled. Thus, for every submitted job we calculate the new run time based on the platform level of the job and decide if it fails or not.

5.3.4 Overhead Model

When the processors have been chosen for the job, the new run time can be calculated. We do this by computing the minimum platform level for the job based on its number of required processors, L_{min}, and the current platform level based on its allocated processors, L_{cur}. Thus, the ratio of slow down applied to the job due to slower than expected communication is $R = L_{cur} - L_{min}$. We model the job's run time depending on the penalty function used. The two penalty functions that are mainly used for the purpose of modifying the job's run time are *linear* and *quadratic*.

For the linear penalty function, the job's new run time is modeled as:

$$T' = T + T \times R \times F, \tag{5.1}$$

where R is the difference between the minimum and current platform levels and F is the penalty factor, a parameter received as input.

For the quadratic penalty function, the job's new run time is:

$$T' = T \times F^R. \tag{5.2}$$

After the job's new run time is know, we verify if the new run time is bigger than the job's wall time. If it is, the new run time is set to be equal the wall time and the job is considered failed.

The advantage of this model is that no information specific to the communication pattern of each job is necessary. The only information needed is the shape and performance of the platform. Other models, like TreeMatch [33], are restricted to jobs that use MPI because they need the communication matrix used by each job.

Additionally, there are some disadvantages to using a communication matrix to describe a job's communication profile. The biggest one is that a communication matrix does not know precisely the size and timestamp of each message, which means it can not diferentiate between two jobs that communicates a lot in a small perior of time and communicates at a constant rate for the duration of the execution.

5.4 Online Scheduling

In an Offline scheduling, the scheduler has complete understanding of the system and the environment it will operate in. More specifically, the scheduler knows, when execution begins, the release date and real run time of all the jobs. With this information, the scheduler is able to backfill jobs with the guarantee that these resources will be available.

Online scheduling works in a different way. Initially, the scheduler has no information about release dates. This means that time advances while the scheduler waits for jobs to be submitted. When a job is submitted, the only two pieces of information about that job the scheduler has and in turn can use are the number of requested processors and user submitted wall time. With this information, the scheduler is able to give the job a reservation, which is composed of a list of processors and a starting time.

When the job starts executing, the scheduler still does not know when it will finish, merely the maximum ending time based on the user submitted wall time. If the execution reaches this wall time, the scheduler is forced to kill the job and consider it failed, as other jobs may have reservations for the same resources later on. Furthermore, if the job finishes before its maximum ending time, the scheduler has the opportunity to re-utilize the resources. This subject is discussed in the next Section.

Figure 5.2 shows an example of a schedule where jobs are being assigned in an Online fashion

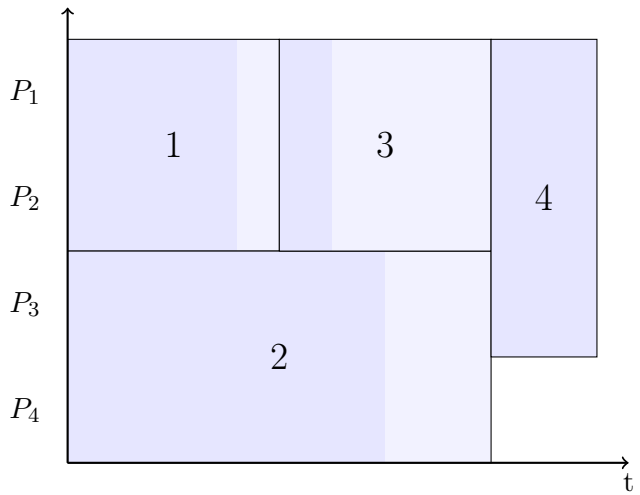


Figure 5.2 – Schedule showing jobs with different user submitted wall times and real run times

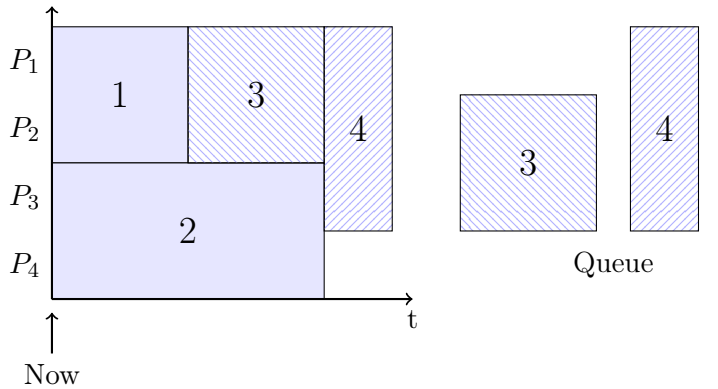
5.5 Job reassignment

One of the things that can be done to further improve the efficiency of the Backfilling algorithm is to reassign jobs that finish their execution early. This is mentioned in Backfilling's original paper but only briefly. In our implementation of the Backfilling algorithm, this is done in the following way:

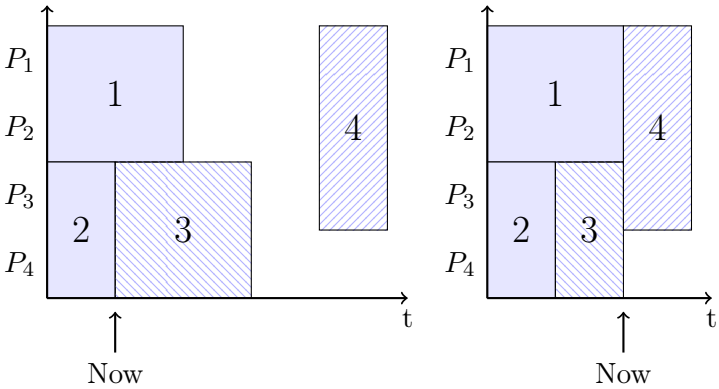
First, it is important to note that jobs receive an assignment that consists of a starting time and a set of processors as soon as they are released. This consists in a reservation and is guaranteed for the job, meaning that no other job can be assigned or backfilled to the time or processors assigned to this job.

When any job that is executing finishes early, the first thing that is done is to release the remaining time and processors that were assigned to this job. After that, the algorithm checks, for every job that is queued, or in other words, that has been assigned but has not started yet, if the job can be backfilled to start now. The reason we only move jobs if they can start now is to avoid all jobs from being reassigned repeatedly every time a job finishes

early. Additionally, reassigning jobs only if they can start now achieves the same result, which is to utilize the platform's resources as much as possible, contributing to a smaller overall flow time.



(a) Schedule when jobs in the queue are assigned



(b) Schedule after job 2 finishes early (c) Schedule after job 3 finishes early

Figure 5.3 – Example of job reassignment

Figure 5.3 illustrate the reassignment process. On Figure 5.3a jobs 1 and 2 have already been assigned and two more jobs, 3 and 4, have been submitted and are being assigned now. This is done in the FCFS fashion. Also, the arrow on the lower part of the Figure indicates the current time. On Figure 5.3b, we can see that job 2 finished earlier than expected. Following that, the jobs that were already assigned but have not started yet can be reassigned. Thus, job 3 is allowed to move forward and start right now. Since job 4 could not start now even though it could move forward, it keeps its original reservation. Finally, Figure 5.3c shows that when job 3 also finishes

early, job 4 is allowed to move forward and start now, instead of waiting for its reservation.

5.6 Allocation Algorithms

The main way we improve the Backfilling algorithm is by implementing *variants*. When Backfilling schedules a job, it chooses the earliest time that have enough processors available for the duration of the job. It does not, however, choose specific processors when more than the minimum necessary are available. That gives us an opportunity to choose the job's processors in a way that improves its execution.

5.6.1 Basic Backfilling

The Basic Backfilling variant try to change the behavior of the Backfilling algorithm as little as possible. The resulting behavior, in this case the set of processors chosen for a particular job, depends a little on the details of the implementation, like data structures and programming language used. Nonetheless, Basic Backfilling chooses the first processors available, as soon as they are selected from the pool of processors. Since in this case processors are kept in a list and this list is ordered, the available processors with the smallest ids are chosen.

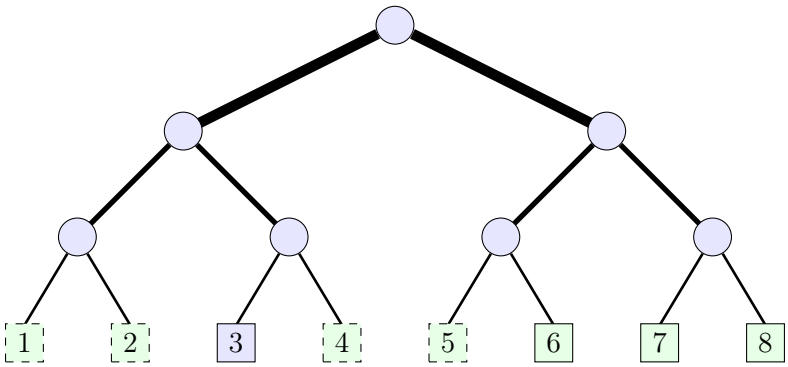


Figure 5.4 – Example of schedule using the Basic Backfilling variant

Figure 5.4 shows an example of a job scheduled using the Basic Backfilling variant. In this case, the first processors available for the job are 1 2, 4 and 5. The Figure also shows an example of a case where picking the first processors available might not be a good idea. The chosen processors are

scattered among different parts of the platform, whereas if processors 5 to 8 were chosen, the assignment would be better in a few different aspects, like lower fragmentation and communication cost between the processors.

5.6.2 Basic Locality

A job is considered *local* if it uses the minimum amount of clusters it needs to execute. This minimum number of clusters is determined by the size of the job. A job that has basic locality has the advantage of keeping the communication between processors contained in the lowest level as much as possible, without any knowledge about the topology of the platform apart from the size of each cluster. An example can be seen on Figure 5.5. A job was allocated and the chosen processors are marked in green. Due to the number of required processors, the job needs two full clusters to be executed. In order for the job to be considered local, any two clusters can be chosen.

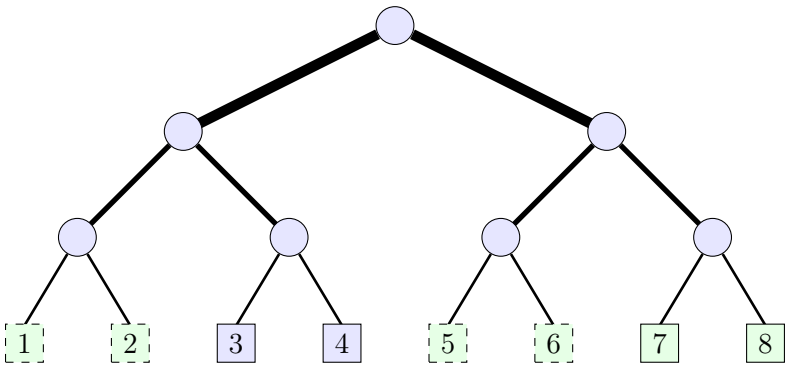


Figure 5.5 – Example of a local job

Best Effort Local

This variant tries to allocate the job in as few clusters as possible. It does that by separating the available processors in blocks pertaining to the clusters and sorting these blocks by descending size. Since this is a *best effort* variant, blocks are selected from the list until enough processors are chosen for executing the job, without delaying it.

Forced Local

This variant does the same thing as *Best Effort Local*, except it is only allowed to select processors from the first element of the sorted list of proces-

sors grouped by clusters. If there are not enough processors for the execution of the job, the job is delayed until the next available time stamp, when a new attempt will be made to allocate it.

5.6.3 Contiguity

We say that a job is *contiguous* if all processors allocated to it form a contiguous range. Contiguous jobs have the advantage of being close to local, using at most one additional cluster over the minimum required. Scheduling contiguous jobs does not require any information about the platform. For this reason, it is a good method of achieving locality when this information can not be used or is not available. An example can be found on Figure 5.6. The processors chosen for the job are marked in green. In order for the job to be considered contiguous, the chosen processors need to form a contiguous range.

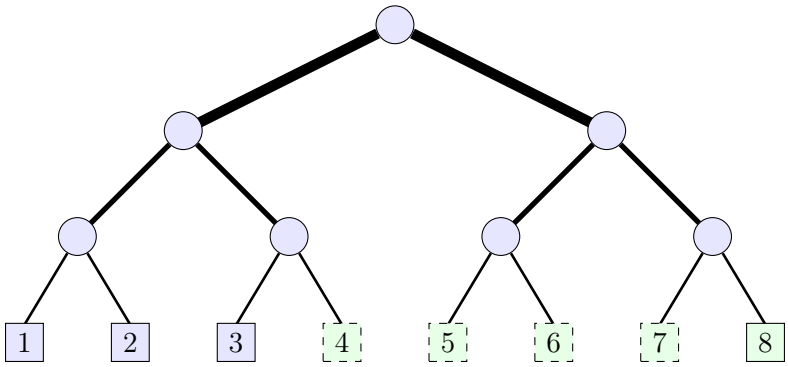


Figure 5.6 – Example of a contiguous job

Best Effort Contiguous

This variant attempts to allocate the job so all chosen processors form a contiguous range. It does that by dividing the list of available processing units in blocks and sorting these blocks by descending size. It then chooses these blocks until enough processors have been chosen. Since this is a *best effort* variant, it does not delay the job. It will choose as many contiguous blocks as it needs until enough processors to execute the job have been chosen.

Forced Contiguous

This variant does the same thing as *Best Effort Contiguous* except it can only use one block of contiguous processors. If the biggest block does not have enough processors for executing the job, the job is delayed and a new attempt will be made at the next available time stamp.

5.6.4 Platform Level (Algorithm 5)

We consider a job's *platform level* the number of different levels of the platform the communication between processors needs to go through to reach any of the other processors. This number is 1 if all the processors allocated to the job are in the same cluster, that is, if they can reach all the other processors going through only one level of the platform. The number is increased for each different level of the platform the communication has to go through.

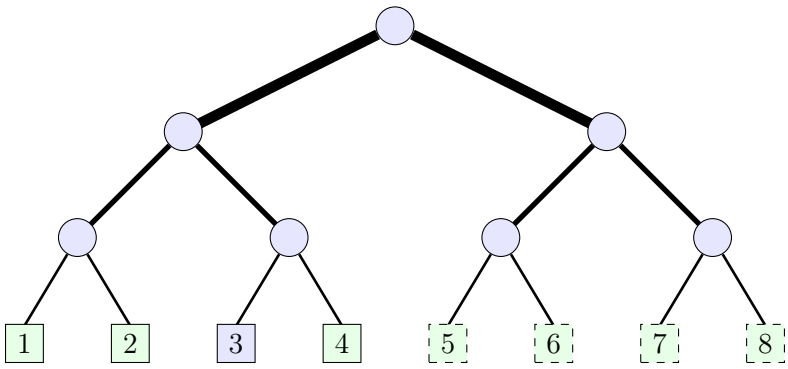


Figure 5.7 – Example of job using the minimum number of platform levels

An example of a job that utilizes the minimum number of platform levels can be seen on Figure 5.7. The chosen processors for the job can be seen marked in green. Since the job needs two clusters, its minimum platform level is 2. Nonetheless, in order for the job to be considered using the minimum number of platform levels, these clusters need to be connected to each other on the next platform level.

Best Effort Platform Level

This variant tries to place the job so that the communication goes though as few levels in the platform as possible without delaying it. It does that by generating a list of platform levels where each item contains all the processors

Algorithm 5: Backfilling with best effort platform

```
Data: job j, set P of available processors, platform topology T
1  foreach topology level l in T do
2      N ← list of nodes on level l;
3      foreach node n in N do
4          P_n ← list of available processors connected to node n;
5          if |P_n| < size_j then
6              continue;
7          end
8          C_n ← array of clusters (initially empty);
9          foreach processor i in C_n do
10             let j be the index of the cluster containing i;
11             assign i to C_n[j];
12         end
13         sort C_n by the number of processors in each cluster in
             decreasing order;
14         return size_j processors of the first clusters in C_n
15     end
16 end
```

that could be used going up to that level. Then, it sorts the elements in the list by descending size. The variant picks the processors from the first level in the list with enough processors for the job.

Forced Platform Level

This variant is similar to *best effort platform* expect it will only allow the first item in the sorted list of platform levels to be used. If this level does not have enough processors for the job, the job is delayed until the next time stamp, when a new attempt can be made.

Discussion

The advantage of using the platform level to choose the processors for a job is that this approach doesn't need any information about the job's communication pattern or composition. That is interesting since getting access to this kind of information is complicated and is only possible for some types of jobs. This approach only needs some information about the

platform, like shape and network profile, which is something that the admins of the platform have easy access to.

Furthermore, this approach focuses on improving the run time of the jobs, which is the most important metric for the user. Results show that improving the platform level of the jobs also improves metrics that may be important for the admins, like flow, stretch, contiguity, job success rate.

5.7 Experimental Results

5.7.1 Goals

The main goal of the experiments is to compare the performance of the different Backfilling variants. Additionally, we want to show that:

- there is considerable performance increase when using a platform aware variant of Backfilling, when comparing to the basic variant;
- that full locality awareness, where jobs are assigned to regions of the platform where the communication has to go through as few levels as possible, is preferable to basic locality awareness, where the platform is split in groups of processors called clusters and jobs are assigned to as few clusters as possible;
- we can improve the basic Backfilling algorithm in different ways without the need for code intrumentalization, without tying to specific runtime environments like MPI and without penalizing jobs with severe delays due to the additional constraints.

5.7.2 Simulator

The Backfilling algorithm and the variants were implemented together with a discrete events simulator written specifically for these experiments. This simulator works by reading a list of jobs from a batch scheduler trace file in the SWF format [9]. During the execution, each job and its associated characteristics are read from the trace. Moreover, each trace is associated with a platform where the jobs were originally executed and running times were measured. Several commonly used platforms and their associated traces were chosen for these experiments.

The reason we chose to implement our own simulator is that since the beginning we wanted to be able to tweak, run experiments and analyze results for several different aspects of batch scheduling and Backfilling, from the

most basic, like changing the priority each job has when it is submitted or choosing on which processors the job is going to be executed, to more complex, like redefining the execution profile and optimizing the code to run the kinds of experiments that we were planning.

In total, the simulator has been worked on for approximately two years, and includes features like:

Implement different variants One of the main goals of the simulator is to help us find different ways to improve Backfilling. For this reason, one of the most important parts of the simulator is the implementation of the variants. These variants inherit the basic implementation of the Backfilling algorithm and make changes to improve it. The biggest change is usually the way processors are chosen, but other, smaller changes can be made as well.

Implement optimized events controller and execution profile One of the reasons why implementing our own simulator is interesting is that we can optimize certain parts of the simulator, like the events controller and the execution profile. This is important because it helps reduce the time it takes to process a trace and assign the jobs, specially for larger traces.

Reassign jobs One of the things that is mentioned in the main Backfilling paper but not explained or experimented upon is the process of reassigning queued jobs when jobs that are executing finish early. This step is very important because it gives Backfilling not just one opportunity to backfill jobs, but many, every time a job finishes early. The downside to doing this is that it increases the complexity of the Backfilling algorithm by one order of magnitude, since every time a job finishes early, all jobs that are queued need to be checked to see if they can be moved forward. This fact makes it more important for the optimization of things like the execution profile, since it is heavily utilized when checking if a job can be backfilled.

Generate traces The simulator allows us to create traces of different sizes and characteristics based on larger, popular traces available in the literature. This is important because original traces tend to be large, spanning several months, which if used directly would take too long to process and wouldn't allow for different instances. Thus, the simulator splits these long traces in segments of approximately seven days and, depending on the size of the platform (number of processors) and the size of the original trace, chooses a

number of contiguous segments to generate a trace that can be used in the experiments.

Prepare experiments Apart from the improvements made to the Backfilling algorithm by implementing our own simulator, one important addition to the simulator is the ability to execute the simulator using the generated instances explained earlier, in addition to a combination of different parameters, and save everything to output files in the CSV (Comma Separated Values) format. These files can then be used to generate the curves included in this text.

5.7.3 Parameters

In our experiments we consider only the real run time of the jobs. This choice is common when doing simulations using real traces since user submitted wall times can be strongly overestimated [51]. The reason for this overestimation is that usually jobs are interrupted and killed if the run time reaches the user submitted wall time.

Our experiments are designed by setting a list of parameters:

Platform Parameters The first set of parameters is related to the platforms. For each platform, we set the number of processors, shape and communication characteristics based on the information available about them in their respective web pages.

Traces Secondly, we prepare the traces associated with each of the platforms. This preparation consists in dividing the traces in periods of seven days. Then, depending on the size of the trace and the platform, we include in the experiments instances with different numbers of periods.

Penalty Thirdly, we configure the different values of penalty function and penalty factor that will be used in the experiments. As explained before, linear and quadratic functions are available.

Instances Finally, we determine how many instances of each trace are used in the experiments. As mentioned before, the original traces are divided in periods of seven days. One instance is a trace containing a contiguous number of seven day periods. The number of periods depends on the size of the platform (i.e number of processors) and the trace itself. This number

of instances is meant to make sure that the results have an acceptable trust interval.

5.7.4 Platforms

Table 5.2 shows the platforms that were included in the experiments, as well as the number of processors, numbers of jobs and numbers of 7 day periods used in each instance.

Platform	**Processors**	**Jobs**	**Periods**
Currie [1]	80640	279991	3
Hpc2n [3]	240	201998	1
Kth-SP2 [10]	100	20483	1
Llnl-Thunder [4]	1024	97875	1
Ricc [7]	8192	431547	2
Sdsc-Blue [8]	1152	195587	1

Table 5.2 – Platforms included in the experiments

5.7.5 Experimental Analysis

The experiments were performed using the following combination of parameters:

- Included platforms, as described by the Table 5.2
- Penality function: quadratic
- Penality factor: 2
- 30 instances of traces from each platform
- Included variants:
  - Basic Backfilling: *basic*
  - Best Effort Contiguity: *becont*
  - Best Effort Locality: *beloc*
  - Best Effort Platform Level: *beplat*
  - Forced Contiguity: *cont*
  - Forced Locality: *loc*

  - Forced Platform Level: *plat*

- Included metrics:
  - Sum flow
  - Sum stretch
  - Contiguity factor
  - Locality factor
  - Platform level factor
  - Job success rate

The machines used to carry out the experiments are part of the Grid5000 testbed for experiment-driven research [2].

Results preparation

After all the instances determined by the set of parameters are executed, the results file is processed and used to generate the graphs shown below. Firstly, the results are grouped by the combination of parameters, so that each group contains the 30 instances for that combination. Then, a normalized value is calculated for each metric in each instance, dividing its value by the average between instances when the variant is Basic. That means that all the values for the other instances are normalized by the Basic Backfilling metric. After that, the average is taken between the different instances, for all the metrics and all the variants. That is the value that is shown in the bar graphs.

Sum Flow

The first Figure we present is Figure 5.8, containing the results for sum flow on the different platforms.

In this Graph and in the following ones, each set of bars corresponds to a different platform included in the experiments. The name of the platform can be found at the top of the graph, shown in grey labels. To the right of the graph another label shows the combination of penalty function and factor used in the experiment. In this case, the quadratic function with a factor of 2 was used. Finally, the legend on the right indicates the different variants that are being compared.

We can see from the Figure that with the exception of Best Effort Local on the SDSC Blue platform, all best effort variants show improvements when

comparing to Basic Backfilling for sum flow. More specifically, we can see the Best Effort Platform shows the best results on all platforms. If we consider the Forced variants, we can see that the Forced Contiguous and Forced Local variants present worst results than Basic Backfilling for the HPC2N platform and Forced Contiguous for the KTH SP2 platform. Other than that, forced variants present improved results when comparing to Basic Backfilling. More specifically, they tend to show results that are similar to their best effort counter part and sometimes even improved results.

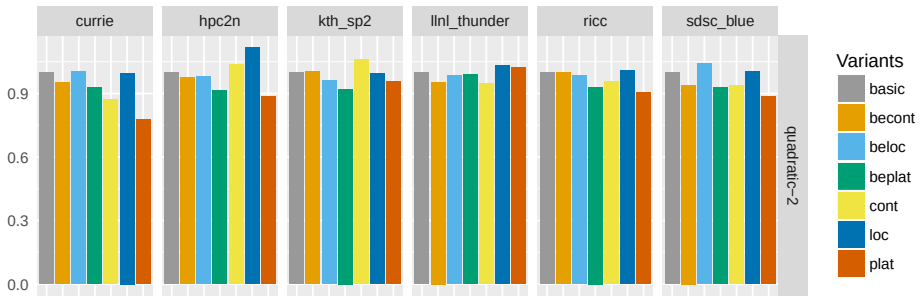
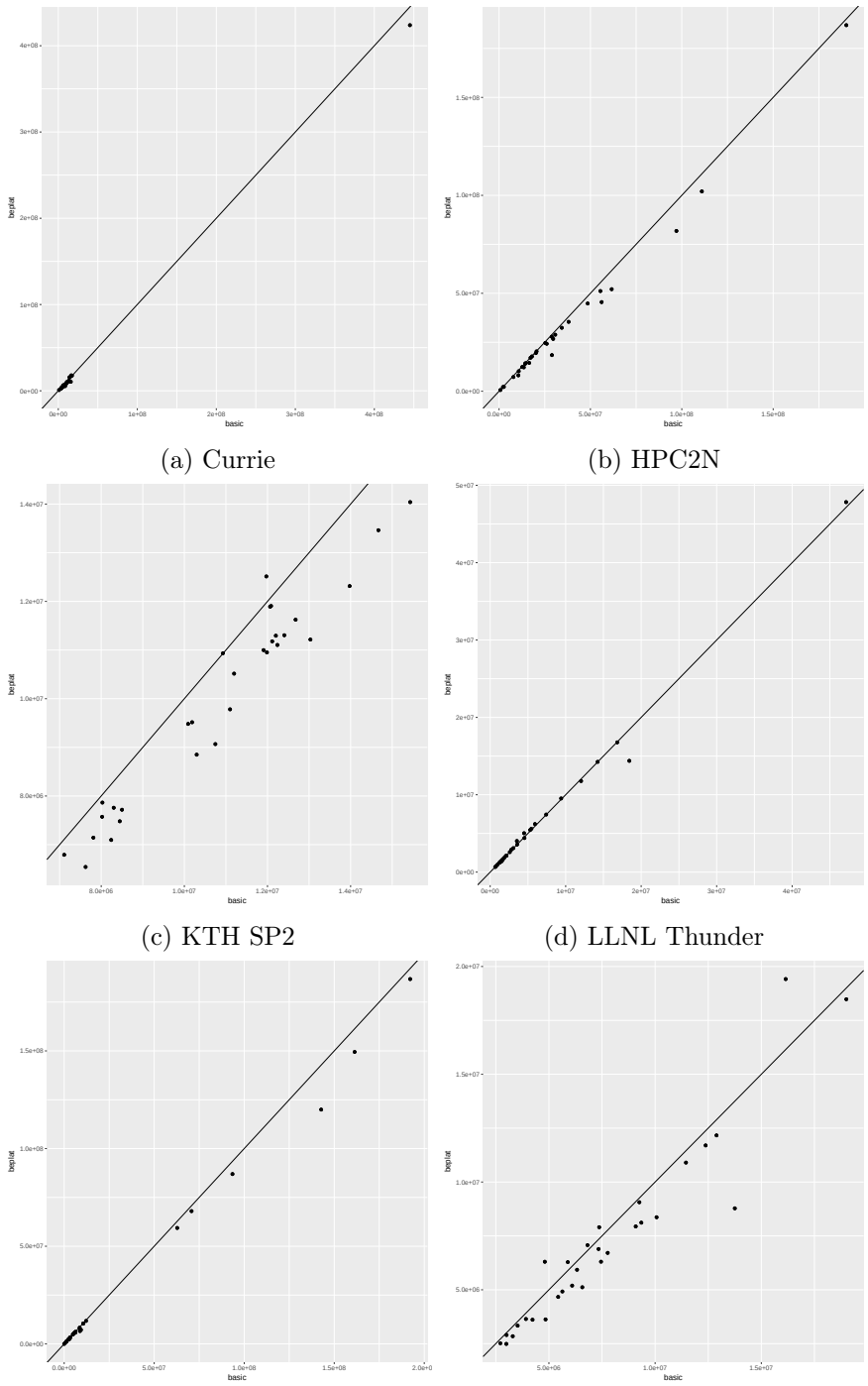


Figure 5.8 – Sum Flow

Next, we present some graphs that aim to take a closer look at the flow metric in some interesting cases. The first set of graphs (Figure 5.9) compares Basic Backfilling to Best Effort Platform for the different platforms included in the experiments. In these graphs, the horizontal axis denotes the sum flow for Basic Backfilling and the vertical axis Best Effort Platform. There is a line indicating the separation between regions where each variant has better results and each point indicates one of the experiment instances. The Figure shows that for all of the included platforms the results are either similar (Currie, LLNL Thunder), marginally better for Best Effort Platform (RICC) or very good for Best Effort Platform (HPC2N, KTH SP2 and SDSC Blue). In these cases, most of the points show a better sum flow for Best Effort Platform.

These points indicate that, because jobs are placed more close together when they are assigned by the Best Effort Platform variant, they tend to finish earlier, opening space in the schedule for other jobs. These other jobs will start earlier and thus have a lower flow time. In other cases, when the flow is mostly similar, jobs either don't finish earlier, not opening space for the next jobs, or they do finish earlier but no jobs can be moved due to their size or not being released yet (not enough jobs).



(a) Currie

(b) HPC2N

(c) KTH SP2

(d) LLNL Thunder

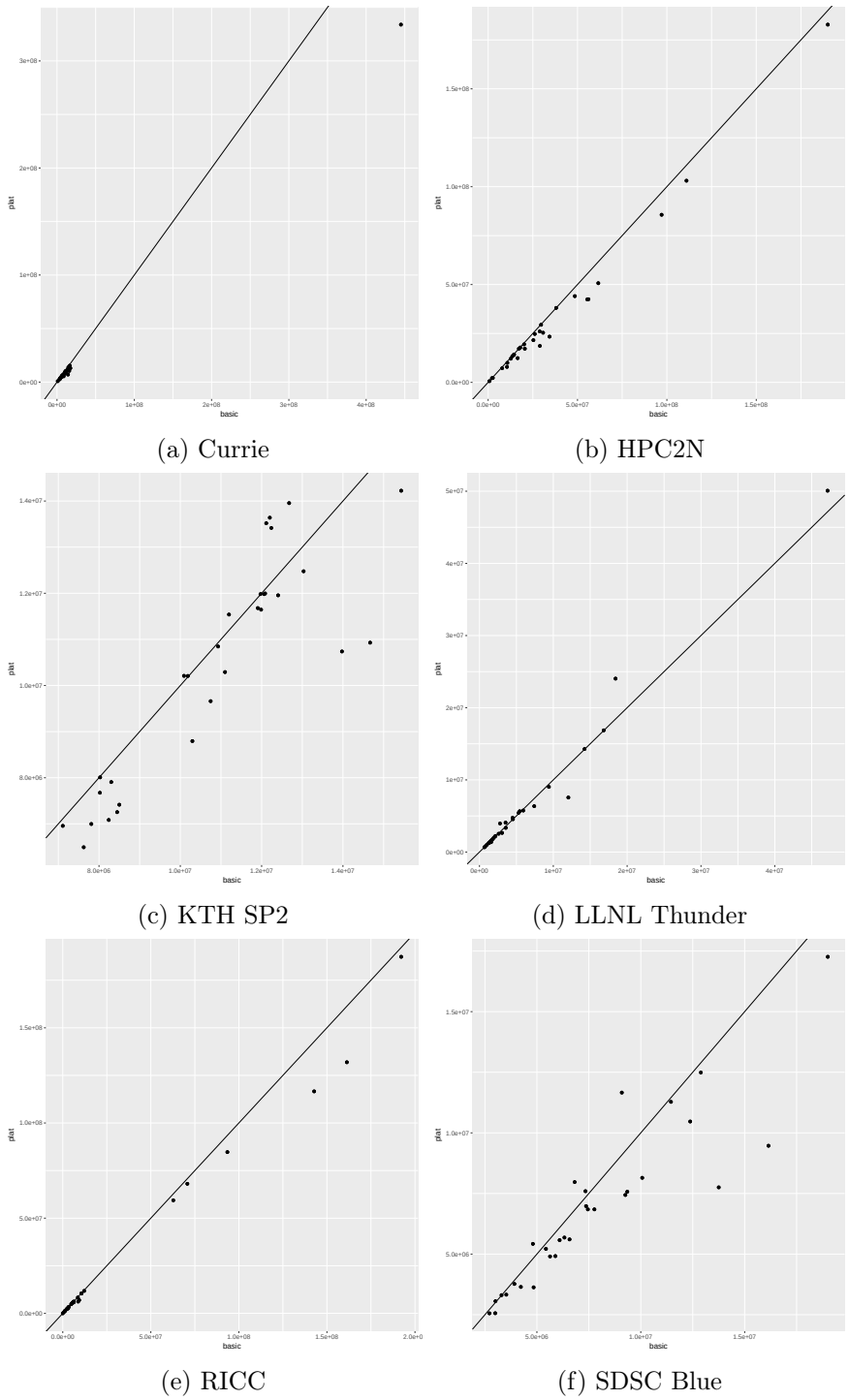
(e) RICC

(f) SDSC Blue

Figure 5.9 – Clouds of points for sum flow comparing the basic and best effort platform variants

Next, we show the set of graphs comparing the Basic Backfilling variant to Forced Platform. Since in this case the constraint is stronger, or in other words, jobs are delayed unless they can be scheduled in the lowest platform level possible, some jobs might be delayed further when comparing to the best effort equivalent. On the other hand, scheduling jobs in this fashion causes them to have the smallest run time possible, helping flow.

As we can see on Figure 5.10, the results when comparing Basic Backfilling to Forced Platform are similar to the ones shown before, even with the stronger constraint. There is improvement on the flow metric for most of the platforms (HPC2N, KTH SP2, RICC and SDSC Blue), while results are similar between the variants for the Currie and LLNL Thunder platform.



(a) Currie

(b) HPC2N

(c) KTH SP2

(d) LLNL Thunder

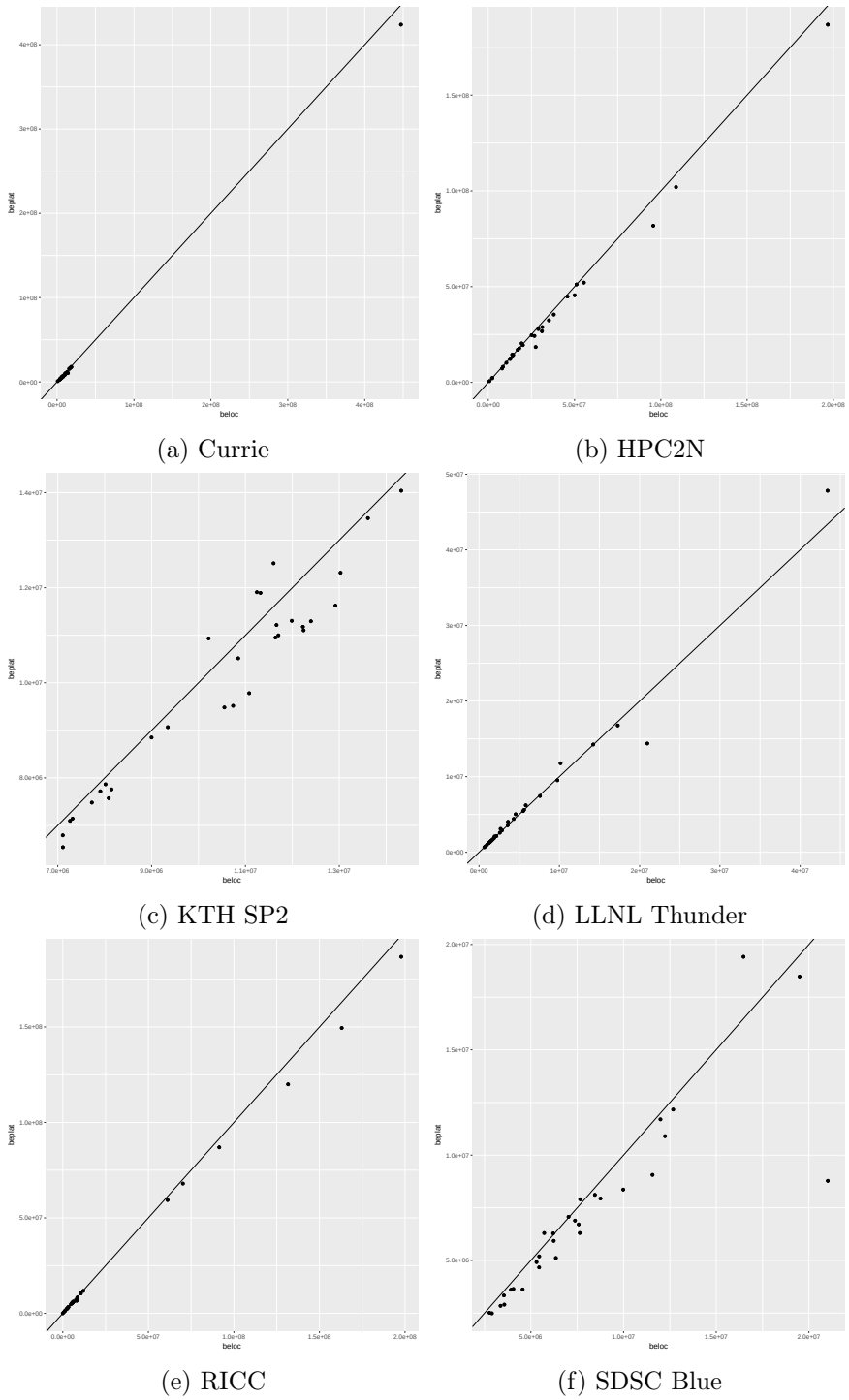
(e) RICC

(f) SDSC Blue

Figure 5.10 – Clouds of points for sum flow comparing the basic and forced platform variants

Next, we show a set of graphs comparing the Best Effort Local, that implements the constraint of simple locality, where the platform is modeled as a set of clusters with no hierarchy, and the Best Effort Platform variants. In this case, results are more similar, as we can expect. More specifically, we can see on Figure 5.11 that results are similar for flow on the Currie, HPC2N and LLNL Thunder platforms, slightly better for the RICC platform and significantly better for the Best Effort Platform variant on the KTH SP2 and SDSC Blue platforms.

What we can understand from these results is that although the basic locality constraint goes towards a better schedule, it is not enough. Jobs might use two clusters and that is considered a good result for the constraint of basic locality, but those clusters might be located far from each other on the platform, leading to a longer run time for the job.



(a) Currie

(b) HPC2N

(c) KTH SP2

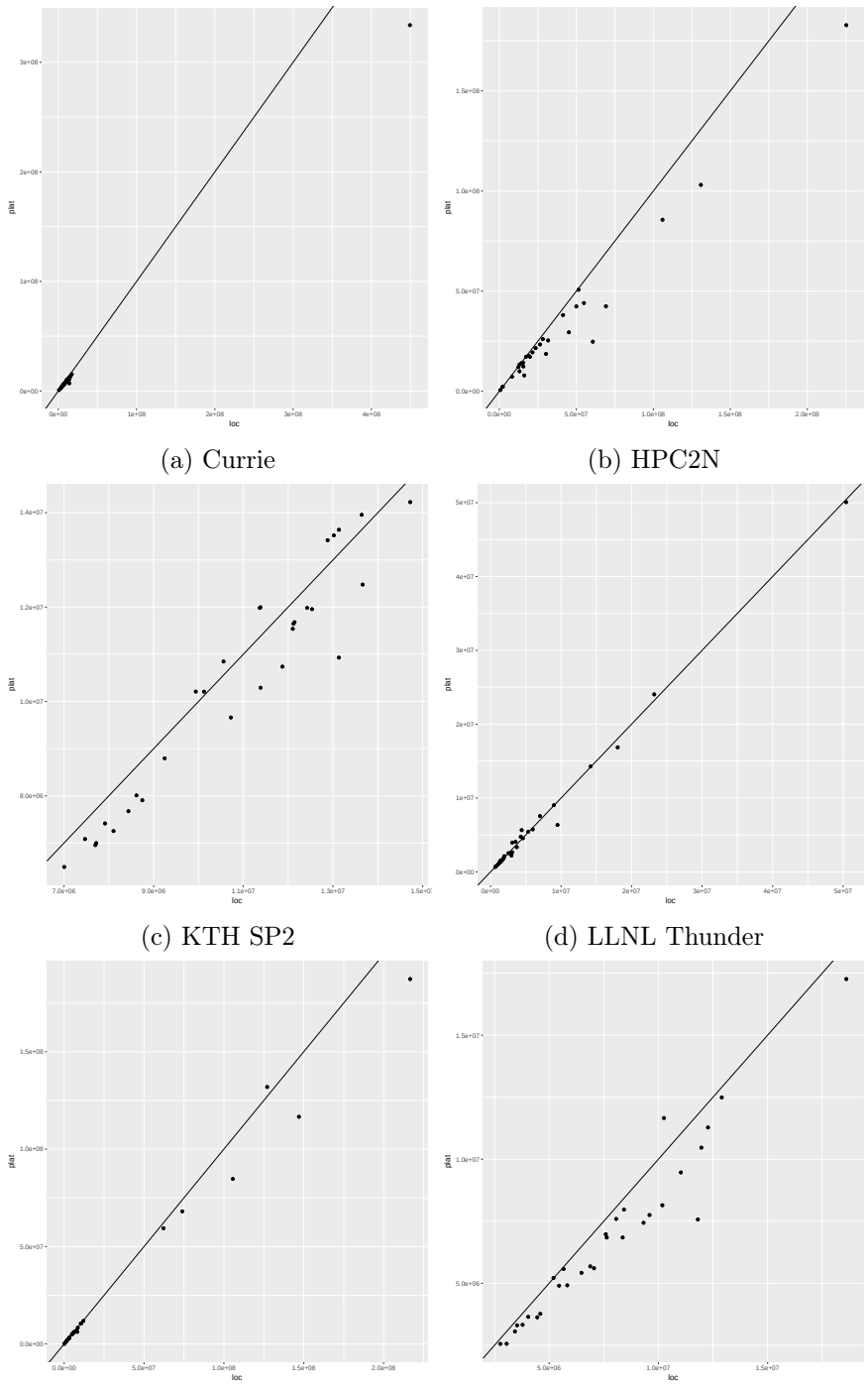
(d) LLNL Thunder

(e) RICC

(f) SDSC Blue

Figure 5.11 – Clouds of points for sum flow comparing the best effort local and best effort platform variants

Finally, we show the set of graphs comparing the Forced Locality variant, that implements basic locality constraint with the difference that it delays jobs unless they use the minimum number of clusters possible, with the Forced Platform variant. Here we see similar results as the previous case. Figure 5.12 shows that on three of the platforms (HPC2N, KTH SP2 and SDSC Blue) results are considerably better for Forced Platform in comparison to Forced Platform. On one of the platforms (RICC) results are slightly better for Forced Platform and for the Currie and LLNL Thunder platforms results are similar.



(a) Currie

(b) HPC2N

(c) KTH SP2

(d) LLNL Thunder

(e) RICC

(f) SDSC Blue

Figure 5.12 – Clouds of points for sum flow comparing the forced local and forced platform variants

Contiguity Factor

The next metric we show is the Contiguity Factor. This metric is calculated by counting how many contiguous blocks have been assigned to a job. After all jobs are assigned, the average Contiguity Factor for the schedule is calculated.

Figure 5.13 shows the results for this metric. We can see in this Figure that basically all variants improve contiguity when comparing to Basic Backfilling, even though, as explained before, this implementation of Backfilling tends to choose contiguous blocks of processing units, if they exist when choosing ranges of processors for a job. This is to be expected, since all variants tend to achieve some level of contiguity, in one fashion or another. Furthermore, we can see that the variants with the best results for this metric are Best Effort Contiguity and Forced Contiguity. This is also expected, since this is desired effect of these variants.

This metric is interesting because, as we will see later on when analyzing the results for Job Success Rate, an improvement in contiguity does help improve the other metrics, although considerably less than other, more locality aware variants. This in addition to the fact that contiguity is easy to implement, doesn't require any information about the platform and is a weaker constraint, meaning that it might impose fewer delays than other variants, makes it interesting.

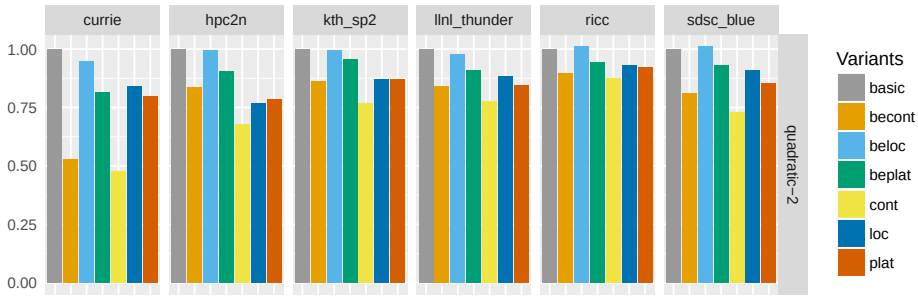


Figure 5.13 – Contiguity Factor

Locality Factor

In this Section, we present and analyze the results for the Locality Factor metric. This metric is calculated by counting how many different clusters have been used in a job's assignment, and dividing that number by the

minimum number of clusters needed for that job, considering how many processors it needs. This number is the job's locality factor.

Figure 5.14 shows the results for this metric. As we can see, all variants improve basic locality when comparing to Basic Backfilling, even Best Effort Contiguous and Forced Contiguous, that don't have any information at all about the platform. Furthermore, the best results are shows by the Best Effort and Forced Locality variants. This is to be expected, since this is precisely what these variants try to improve. Finally, we see that the Best Effort and Forced Platform variants achieve very good basic locality, even though they use a different model and try to achieve a different goal.

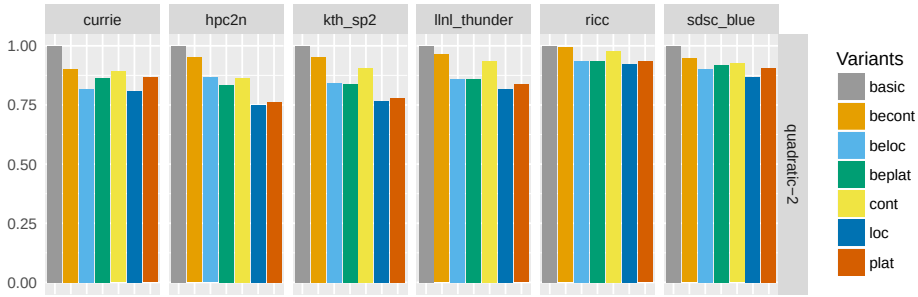


Figure 5.14 – Locality Factor

Platform Level Factor

In this Section, we present and analyze the results for the Platform Level Factor metric. This metric is calculated by counting, for a job, how many levels the communication has to go through to reach all the processors assigned to that job. Then, that number is divided by the minimum number of levels the communication has to go through, considering the number of processors required by the job.

Figure 5.15 shows the results for this metric. We can see that all the metrics have better platform level factors in comparison to Basic Backfilling. Additionally, it is interesting to note that for this metric, the stronger the constraint, the better. That can be explained to the fact that stronger constraints tend to pack jobs closer together, increasing the probability that those jobs are assigned to lower levels of the platform. Furthermore, we can see that the best results are shown by the Best Effort and Forced Platform variants. This means that when utilizing these variants jobs can communicate between their processors going through fewer levels of the platform,

contributing smaller run times.

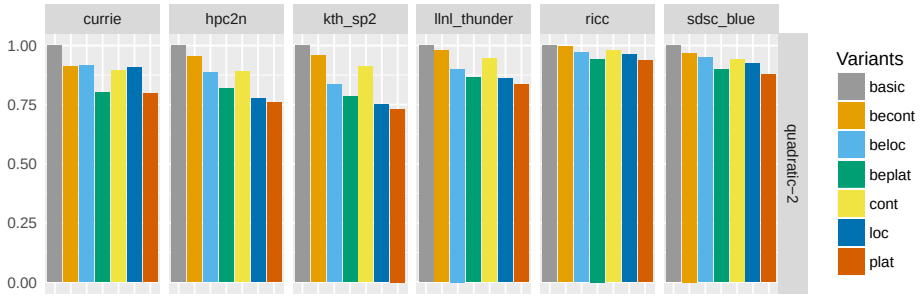


Figure 5.15 – Platform Level Factor

Job Success Rate

Lastly, we show the results for the Job Success Rate metric. This metric is calculated by counting how many jobs are successfully executed in a particular schedule or trace. The more jobs that are successfully executed, the better.

Figure 5.16 shows the results for the metric. As with the Platform Level Factor metric, all variants present improved results when comparing to Basic Backfilling. Indeed, this is expected since there is a direct relationship between these two metrics, meaning that the platform level of a job directly impacts its run time, which in turn decides if the job is successful or not in its execution. Thus, we can see that the best results are the ones shown by the Best Effort and Forced Platform variants, for all the platforms.

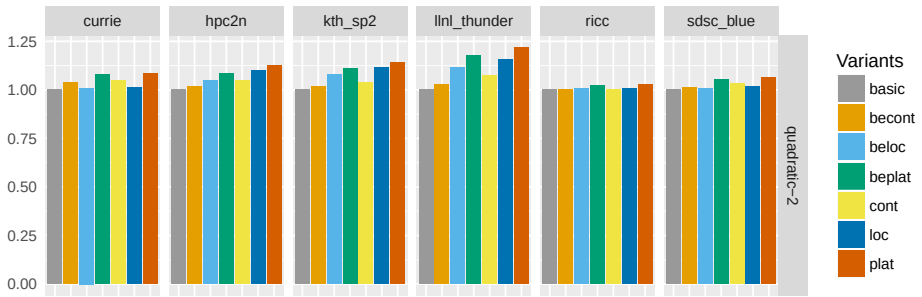


Figure 5.16 – Job Success Rate

5.8 Conclusion

In this Chapter, we update our simulator to include job reassignment, which is done when jobs that are running finish early, and online scheduling. In online scheduling, the scheduler does not know the real run time of the jobs and instead relies on user submitted wall times to assign jobs.

In our experiment campaign, we show that the Full Locality constraint is superior to all the previously proposed constraint, and specially Basic Backfilling. The comparison is done on several commonly used platforms, using a combination of generated instances and parameters. The results are presented with different metrics, demonstrating that our proposed constraint improves the time jobs stay in the system while simultaneously reducing fragmentation, which allows for a better utilization of the platform. These improvements contribute to an overall better job success rate.

Chapter 6

Conclusion

In this work, we focused our study on designing efficient resource allocation for parallel systems. At a lower level, this corresponds to parallelize a job (composed of several tasks), given certain resources, that may include various types of processing units, should try to match the characteristics of each task to the resources, utilizing them as efficiently as possible. At a higher level of the whole platform, we focused our studies on studying efficient methods for assigning multiple jobs to resources on a platform where locality matters. This means that the platform has a hierarchical structure where an assignment in which the assigned processors are close together is advantageous when comparing to a similar assignment containing processors that are far apart from each others. Both situations involve a variant of allocation and scheduling problems. At the lower level, the scenario can involve multi-processor machines, different levels of cache while at a higher level, the hierarchy indicates clusters that can be geographically separated. As a consequence, the network imposes a certain latency on the communication.

In the first Chapter, we were interested in showing how tasks can be assigned to different types of processing units in order to utilize available resources as efficiently as possible. To achieve this goal, we proposed an efficient implementation of the classical Smith-Waterman algorithm utilizing SIMD and SIMT parallelization schemes. Additionally, we employed a novel dual approximation technique in combination with the standard master-slave model for better resource utilization. When comparing query sequences to a common bio-computing benchmark (the UniProt database). A speed of 225 billion cell updates per second (GCUPS) was achieved on a dual Intel Xeon processor system with Nvidia Tesla GPUs, reducing significantly the execution times. In addition to that, the combination of GPUs and CPUs

reduced drastically the execution time for that database, which was faster than all the compared implementations.

In the second Chapter of the thesis, we studied the effect that the combination of mixed jobs with both long and short running times have on the overall flow time. We presented a method for on-line scheduling of independent tasks on a multi-processor machine based on the characterization of a small number of tasks as heavy. Those tasks are considered problematic for the whole performance of the system. We proposed deterministic random approaches for detecting the heavy tasks. These methods were evaluated on instances extracted by seven real traces. In general, we found out that the deterministic method outperforms the standard Shortest First (SPT) policy and in many cases its performance approached the performance of the preemptive policy SRPT which can be considered as a lower bound.

In the third Chapter, we focused our studies on how to schedule jobs utilizing contiguity and locality constraints. We provided a theoretical analysis of the impact of these constraints on the completion time. We presented two different ways to adapt the allocation step of the standard First Come First Come (FCFS) with the Backfilling algorithm to contiguity and locality. More specifically, we distinguished the proposed algorithms between best effort and a more strict enforcing. Our simulation campaign showed that the proposed algorithms did not affect the makespan in a negative way. We were able to improve locality with especially strong improvement for forced contiguous, best effort local and forced local. Experiments showed that very simple constraints can indeed achieve a more local job allocation with little or even no information on platform topology as in the case of forced contiguous.

In the fourth and last Chapter, we updated our simulator to include job reassignment, which is done when jobs that are running finish early, and online scheduling. In online scheduling, the scheduler does not know in advance the real running time of the jobs and instead relies on user submitted wall times to assign jobs. In our experimental campaign, we showed that the Full Locality constraint strategy is superior to all the previously proposed constraints, and specially Basic Backfilling. The comparison was done on several commonly used platforms, utilizing a combination of generated instances and parameters. The results are presented with different metrics, demonstrating that our proposed constraint was able to improve the time jobs stay in the system while simultaneously reducing fragmentation, which allowed for a better utilization of the platform. These improvements contribute to an overall better job success rate.

Bibliography

[1] The cea curie log. http://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie/index.html. Accessed: 2016-11-29.

[2] Grid'5000 testbed. http://grid5000.fr. Accessed: 2017-03-01.

[3] The hpc2n seth log. http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html. Accessed: 2016-11-29.

[4] The llnl thunder log. http://www.cs.huji.ac.il/labs/parallel/workload/l_llnl_thunder/index.html. Accessed: 2016-11-29.

[5] Maui, moab, torque. http://www.adaptivecomputing.com/.

[6] The metacentrum log. http://www.cs.huji.ac.il/labs/parallel/workload/l_metacentrum/index.html. Accessed: 2016-11-29.

[7] The ricc log. http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html. Accessed: 2016-11-29.

[8] The san diego supercomputer center (sdsc) blue horizon log. http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/index.html. Accessed: 2016-11-29.

[9] The standard workload format. http://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

[10] The swedish royal institute of technology (kth) ibm sp2 log. http://www.cs.huji.ac.il/labs/parallel/workload/l_kth_sp2/index.html. Accessed: 2016-11-29.

[11] Tgcc curie. http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm.

[12] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[13] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998.

[14] M. Bianco and B. Cumming. A generic strategy for multi-stage stencils. In *20th International Conference on Parallel Processing (Euro-Par)*, volume 8632 of *LNCS*, pages 584–595. Springer, 2014.

[15] I. Bładek, M. Drozdowski, F. Guinand, and X. Schepler. On contiguous and non-contiguous parallel task scheduling. Technical Report RA-6/2013, Institute of Computing Science, Poznań University of Technology, 2013.

[16] Azzedine Boukerche, Alba Cristina Magalhaes Alves de Melo, Edans Flavius de Oliveira Sandes, and Mauricio Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007.

[17] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, pages 84–93, 2001.

[18] Chunxi Chen and Bertil Schmidt. An adaptive grid implementation of dna sequence alignment. *Future Generation Computer Systems*, 21(7):988–1003, 2005.

[19] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[20] Edans Flavius de O Sandes and Alba Cristina Magalhaes Alves de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211. IEEE, 2011.

[21] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[22] Maciej Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[23] Pierre-François Dutot, Erik Saule, Abhinav Srivastav, and Denis Trystram. Online non-preemptive scheduling to optimize max stretch on a single machine. In *International Conference on Computing and Combinatorics (COCOON)*, volume 9797 of *LNCS*, pages 483–495. Springer, 2016.

[24] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[25] D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *International Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, volume 1291 of *LNCS*, pages 238–261. Springer, 1997.

[26] Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 2221 of *LNCS*, pages 188–206. Springer, 2001.

[27] R. Gibbons. A historical application profiler for use by parallel schedulers. In *International Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, volume 1291 of *LNCS*, pages 58–77. Springer, 1997.

[28] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[29] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[30] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[31] S. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In *International Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, volume 1162 of *LNCS*, pages 27–40. Springer, 1996.

[32] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the Maui scheduler. In *7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 2221 of *LNCS*, pages 87–102. Springer, 2001.

[33] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. Communication and Topology-aware Load Balancing in Charm++ with TreeMatch. September 2013.

[34] Xianyang Jiang, Xinchun Liu, Lin Xu, Peiheng Zhang, and Ninghui Sun. A reconfigurable accelerator for smith–waterman algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(12):1077–1081, 2007.

[35] James Patton Jones and Bill Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 1999.

[36] Akihisa Kako, Takao Ono, Tomio Hirata, and Magnús M Halldórsson. Approximation algorithms for the weighted independent set problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 341–350. Springer, 2005.

[37] Safia Kedad-Sidhoum, Fernando Mendonca, Florence Monna, Gregory Mounié, and Denis Trystram. Fast biological sequence comparison on hybrid platforms. *International Conference on Parallel Processing*, 2014.

[38] Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling independent tasks on multi-cores with gpu accelerators. In *In 11th HeteroPar 2013, International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, in conjuction with the Euro-Par 2013 conference*, Aachen, Germany, Aug 2013.

[39] Hans Kellerer, Thomas Tautenhahn, and Gerhard J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM J. Comput.*, 28(4):1155–1166, 1999.

[40] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. *J. Scheduling*, 11(5):381–404, 2008.

[41] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, 73:875–891, 2007.

[42] V. J. Leung, D. P. Bunde, J. Ebbers, S. P. Feer, N. W. Price, Z. D. Rhodes, and M. Swank. Task mapping stencil computations for non-

contiguous allocations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 377–378, 2014.

[43] D. A. Lifka. The ANL/IBM SP scheduling system. In *International Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, volume 949 of *LNCS*, pages 295–303. Springer, 1995.

[44] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC research notes*, 3(1):93, 2010.

[45] Giorgio Lucarelli, Fernando Mendonca, and Denis Trystam. A new online method for scheduling independent tasks. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017.

[46] Giorgio Lucarelli, Fernando Mendonca, Denis Trystam, and Frederic Wagner. Contiguity and locality in backfilling scheduling. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[47] Giorgio Lucarelli, Nguyen Kim Thang, Abhinav Srivastav, and Denis Trystram. Online non-preemptive scheduling in a resource augmentation model based on duality. In *European Symposium on Algorithms (ESA)*, volume 57 of *LIPIcs*, pages 63:1–63:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[48] Silviano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1st edition, 1990. Wiley Series in Discrete Mathematics and Optimization.

[49] Xiandong Meng and Vipin Chaudhary. A high-performance heterogeneous computing platform for biological sequence analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 21(9):1267–1280, 2010.

[50] David W Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004.

[51] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[52] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes Gehrke. Online scheduling to minimize average stretch. *SIAM J. Comput.*, 34(2):433–452, 2004.

[53] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso. Effects of topology-aware allocation policies on scheduling performance. In *14th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 5798 of *LNCS*, pages 138–156. Springer, 2009.

[54] William R Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in enzymology*, 183:63–98, 1990.

[55] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *Parallel and Distributed Systems, IEEE Transactions on*, 15(12):1070–1081, 2004.

[56] J. Remy. Resource constrained scheduling on multiple machines. *Information Processing Letters*, 91(4):177–182, 2004.

[57] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011.

[58] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kalé. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 807–818. IEEE, 2014.

[59] Aarti Singh, Chen Chen, Weiguo Liu, Wayne Mitchell, and Bertil Schmidt. A hybrid computational grid architecture for comparative genomics. *Information Technology in Biomedicine, IEEE Transactions on*, 12(2):218–225, 2008.

[60] Jaideeep Singh and Ipseeta Aruni. Accelerating smith-waterman on heterogeneous cpu-gpu systems. In *Bioinformatics and Biomedical Engineering,(iCBBE) 2011 5th International Conference on*, pages 1–4. IEEE, 2011.

[61] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - loadleveler API project. In *International Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, volume 1162 of *LNCS*, pages 41–47. Springer, 1996.

[62] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[63] Wayne E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956.

[64] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. Swps3–fast multi-threaded vectorized smith-waterman for ibm cell/be and× 86/sse2. *BMC Research Notes*, 1(1):107, 2008.

[65] J. Tao and T. Liu. WSPT's competitive performance for minimizing the total weighted flow time: From single to parallel machines. *Mathematical Problems in Engineering*, 10.1155/2013/343287, 2013.

[66] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 2862 of *LNCS*, pages 44–60. Springer, 2003.