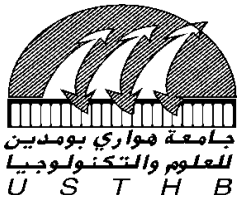
N° d'ordre :01/2018-D/INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediène

Faculté d'electronique et d'informatique
Département d'informatique



THESE
Doctorat en Sciences
Présentée pour l'obtention du **grade** de **Docteur**
En : Informatique
Spécialité : Intelligence artificielle et Ingénierie de logiciels

Par : **DABAH Adel**

Thème

Méthodes de résolution parallèles sur environnement hybride, **Multi-Core** *et GPU pour le problème job shop avec la contrainte de blocage*.

Soutenue publiquement, le 19/06/2018 à 9h30, devant le jury composé de :

Mme	Dalila Boughaci	Professeur	USTHB	Présidente
M.	Abdelhakim AitZai	M.C.A	USTHB	Directeur de thèse
M.	Ahcène Bendjoudi	M.R.A	CERIST	Co-Directeur de thèse
Mme	Fatima Benbouzid	Professeur	ESI,	Examinatrice
M.	Mourad Boudhar	Professeur	USTHB	Examinateur
M.	Yahia Lebbah	Professeur	Univ-Oran1	Examinateur
M.	Riyadh Baghdadi	Docteur	M.I.T.USA	Invité

Thesis advisor: AIT Zai Abdelhakim DABAH Adel

Hybrid CPU/GPU Parallelization For Solving the Blocking Job Shop Scheduling Problem.

Abstract

The classical job shop scheduling problem (JSSP) consists to schedule the execution of a set of jobs on a set of machines while respecting the different constraints. The job shop problem with blocking constraint is a variant of the classic job shop scheduling problem where machines have limited or no storage capacity. Thus, a job must wait on the current machine until its next machine becomes available for processing. The classical JSSP is known to be NP-hard and its blocking extension is even more difficult to solve. Several methods have been proposed to solve this problem (Branch and Bound algorithm, Tabu search, genetic algorithms, etc.), However, they are greedy in terms of computing time for large problem instances, due to their huge search space. Parallel computing architectures, such as multi-core CPUS, GPUs, and clusters, represent an efficient way to deal with such instances. The challenge is to exploit all levels of parallelism by reviewing the design and implementation of these algorithms. In order to effectively solve this problem, we propose in this thesis new efficient hybrid parallel schemes that exploit heterogeneous computing architectures for both exact and approximate resolution methods. Experiments using reference benchmark instances show the efficiency of our hybrid parallel approaches in terms of both the execution time and the solution quality. Moreover, we have been able to solve optimally ten benchmark instances for the first time in the literature and improve the best-known solutions for more than fifty benchmark instances.

Thesis advisor: AIT Zai Abdelhakim DABAH Adel

Résumé

Le problème classique d'ordonnancement d'atelier Job Shop (JSSP) consiste à planifier l'exécution d'un ensemble de travaux (jobs) sur un ensemble de machines tout en respectant les différentes contraintes et en optimisant un ou plusieurs objectifs. Le problème d'ordonnancement job shop avec la contrainte de blocage est une variante du problème JSSP classique obtenue en réduisant ou bien supprimant la capacité de stockage entre les machines. Ainsi, un job doit attendre sur la machine en cours jusqu'à ce que la machine suivante, sur laquelle il va s'exécuter, soit disponible pour le traitement. Ce problème est connu d'être NP-complet au sens fort. Due à l'impact économique de cette contrainte, plusieurs auteurs ont proposé la résolution de ce problème en utilisant les méthodes exactes et les méthodes approchées. Cependant, ces méthodes de résolution sont gourmandes en termes de temps de calcul pour les instances de grande taille. Le calcul parallèle fournit à travers des plates-formes informatiques hétérogènes un moyen efficace de traiter de telles instances. Le défi consiste à exploiter tous les niveaux du parallélisme et revoir la conception et la mise en œuvre de ces algorithmes. Afin de résoudre efficacement ce problème, nous proposons dans cette thèse de nouveaux schémas de parallélisation efficaces qui exploitent des architectures de calcul hétérogènes pour l'accélération des méthodes de résolution exactes et approchées. Les expérimentations numériques en utilisant des benchmarks de référence montrent l'impact positif du parallélisme sur l'amélioration du temps d'exécution et de la qualité des solutions. De plus, nous avons pu résoudre de façon optimale dix instances jamais résolues dans la littérature grâce à nos méthodes de résolution exactes et d'améliorer les meilleures solutions connues dans la littérature pour plus de 50 benchmarks grâce à nos méthodes approchées.

Contents

Listing of figures

LIST OF PUBLICATIONS

The work of this thesis is based on the following publications:

- Adel Dabah, Ahcène Bendjoudi, Abdelhakim AitZai, Didier El-Baz, and Nadia Nouali Taboudjemat. Hybrid multi-core CPU and GPU-based B&B approaches for the blocking job shop scheduling problem. *Journal of Parallel and Distributed Computing*, Volume 117, Pages 73-86, 2018.

- Adel Dabah, Ahcene Bendjoudi, and Abdelhakim AitZai. An efficient tabu search neighborhood based on reconstruction strategy to solve the blocking job shop scheduling problem. *Journal of Industrial & Management Optimization*, 13(4):2015–2031, 2017.

- Adel Dabah, Ahcene Bendjoudi, and Abdelhakim AitZai. Efficient parallel tabu search algorithm for the blocking job shop scheduling problem. Under review for the journal of Soft Computing, Springer 2018.

- Adel Dabah, Ahcène Bendjoudi, Didier El-Baz, Abdelhakim Aitzai. Gpu-based two level parallel B&B for the blocking job shop scheduling problem. In *Parallel and Distributed Processing Symposium Workshops*, 2016 IEEE International, pages 747–755. IEEE, 2016.

- Adel Dabah, Ahcene Bendjoudi, and Abdelhakim Ait Zai. Efficient parallel B&B method for the blocking job shop scheduling problem. In *High Performance Computing & Simulation (HPCS)*, 2016 International Conference on, pages 784–791. IEEE International Conference, 2016.

- Adel Dabah, Ahcène Bendjoudi, Abdelhakim Ait Zai, Didier El-Baz, and Nadia Nouali Taboudjemat. Multi and many-core parallel B&B approaches for the blocking job shop scheduling problem. In *High Performance Computing & Simulation (HPCS)*, 2016 International Conference on, pages 705–712. IEEE, 2016.

Acknowledgments

First of all, I praise Allah Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study successfully.

Then, I warmly thank my dear parents and sisters who have put at my disposal all that is needed (love, support, encouragement, means, etc.), since my first day of school until this day.

I would like to thank very much my thesis adviser Ait Zai Abdelhakim for his trust, availability, advice and his support. During this thesis, I learned a lot from him; he represents for me a teacher, as well as a friend, an inspiration, and a model.

I would like to thank also my co-adviser Bendjoudi Ahcene for all his advice, his availability, and his support during this thesis.

I would like to thank the jury members: Professor Dalila Boughaci from the Department of Computer Science of the USTHB university, Professor Fatima SiTayeb from the Higher School of Computer Science (ESI), Professor Morad Boudhar from the Faculty of Mathematics of the USTHB university, and Finally, Professor Yahia Lebbah from the Computer Science Department of the University of Oran1, and Doctor Riyadh Baghdadi from Massachusetts Institute of Technology (MIT) USA for their time and effort spent in handling this thesis.

I wish to express my gratitude to Dr. Djamal Belazzougui for all his time and effort in improving this thesis, his valuable help, his ideas, in addition to his availability to all my requests.

I would like to thank Mrs Nouali Nadia, Said Yahiaoui, and Ahcene Bendjoudi for their help and support.

I would like to thank warmly all my colleagues at CERIST for all their help and support.

Finally, I warmly thank all my friends as well as anyone who helped me in one way or another in carrying out this work.

1 Introduction

In general, a scheduling problem aims to schedule a set of jobs on a set of resources aiming to achieve some goals. These goals are part of our everyday's life, for example, transportation, manufacturing, projects, budget, etc. Most of these problems are extremely complex and solving them is a time-consuming operation. One of the best ways to deal with such a challenge is to use High-Performance Computing Architectures.

A few years ago, parallelism and high computing architectures where associated with critical areas such as space exploration and defense. Nowadays, most of the computers are parallel and have become more and more advanced and affordable. This fact allows researchers to take benefit from the opportunities offered by the parallelism for the resolution of various problems.

Scheduling problems are encountered in multiple areas including both academic and industrial fields. They still represent a major challenge for researchers due to their complexity which is NP-hard in most cases. Solving these problems can be done using exact or approximate methods. Exact methods aim to solve optimally an optimization problem by exploring the whole search space which is a time consuming operation, especially for large problem

instances. The need of finding quickly acceptable solutions in reasonable time, for many of these problems, has led to the development of approximation algorithms. According to many researchers, exact and approximate methods have proven their utility in solving various kind of combinatorial optimization problems. The success of these methods is explained essentially by their adaptability to different constraints related to real-world problems and by their ease of implementation. The field of combinatorial optimization is still relevant and many efforts are currently being made to improve the performance of the resolution methods. With the evolution of computers and the accessibility of High Performance Computing (HPC) architectures at relatively low cost, many authors see the parallel computing as a very promising way for improving the performance of resolution methods. However, the effects of parallelism and the behavior of parallel resolution methods are still poorly studied, especially for metaheuristics.

The general objectives of our thesis work can be summarized in two points. The first one is to study the relationships between parallel computing and resolution methods in order to better understand the benefits that parallelism can bring for solving combinatorial optimization problems. The second one is to design, identify, and implement efficient parallel solutions to a practical combinatorial optimization problem, namely, the Blocking Job Shop Scheduling (BJSS) problem. This problem is encountered in multiple areas including industrial manufacturing with no storage space, transportation, etc. The achievement of these goals then passes through an adequate knowledge of parallelism, exact and approximate resolution methods, and scheduling problems.

In Chapter 2, we introduce the necessary background information related to scheduling problems in general in addition to a detailed description of the problem treated in this thesis named "The Blocking Job Shop Scheduling Problem (BJSS)". This problem is known to be NP-hard in the strong sense and its search space is equal to $((\textit{number of jobs})!)^{\textit{number of machines}}$ [13]. Due to the importance of the blocking constraint and its huge economic impact, the BJSS problem is relatively well studied in the literature. The first part of this chapter describes the basic elements of scheduling problems which are jobs, machines, constraints, and objectives. Due to the existence of an uncountable number of scheduling problems, we explain the $\alpha|\beta|\gamma$ notation used to classify them in addition to some practical examples of this notation. Another important classification approach is the one that aims to classify the problems according to the space and time required to solve them. The main tool used for this classification is the reduction which is a very important concept aiming to deduce the

complexity of a problem by using another problem whose complexity is known in advance. After that, we give a brief introduction to some of the most widely used resolution methods to solve optimization problems. The second part of this chapter describes in detail the blocking job shop scheduling problem and its applications in real-world problems.

In Chapter 3, we give a general introduction to parallelism and parallel computing architectures and their classifications. Moreover, this chapter contains taxonomies of parallel resolution methods. Indeed, several authors have made a detailed study on the parallelization of exact and approximate resolution methods aiming at exploiting efficiently the different computing architectures in order to face the complexity of combinatorial optimization problems. At the end of this chapter, a detailed related work section about solving the BJSS problem and the parallelization of exact and approximate resolution methods is given.

In Chapter 4, we investigate the exact resolution of the BJSS problem by the means of the Branch and Bound algorithms (B&B). The B&B algorithms are well-known techniques for solving optimally optimization problems. They consist in exploring all feasible solutions in a smart way by avoiding the exploration of unpromising branches. As already mentioned, the BJSS problem is known to be NP-hard. To illustrate the BJSS problem complexity, we show a small example with ten jobs and ten machines. For such example, we have approximately $3,959 \times 10^{63}$ possible solutions. Exploring all these possibilities by the B&B algorithm in order to report the best one takes an unacceptable running time. To deal with this challenge, we propose in this chapter new parallel B&B schemes that exploit several HPC architectures. The first section of this chapter explains our proposed serial B&B algorithm and its components. The second section of this chapter focuses on reducing the B&B execution time by parallelizing the bounding phase of the algorithm using Graphical Processing Units (GPUs) as massively parallel machines. This parallelization is motivated by the fact that this phase consumes more than 90% of the sequential execution time. Two parallel schemes are proposed in this section. The first scheme is a node-based parallelization exploiting the idea that the evaluation (bounding) of each node of the search tree can be calculated in parallel on the GPU. While the second scheme generalizes the idea of the first scheme to increase the ratio of computation to communication on the GPU. The third section of this chapter describes our Multi-core CPU B&B scheme dedicated to personal computers as well as cluster-based architectures. The proposed scheme is based on the parallel search tree exploration (tree based parallelization) using the Master/Worker paradigm. In this scheme, the master performs a breadth-first exploration to ensure wide availability of sub-problems and the workers

perform a depth-first exploration in order to browse a large number of feasible solutions; thereby, obtaining a faster improvement of the upper bound. Our proposed scheme can serve both for exact and approximate approaches at the same time by reporting the best-explored solution during 7,200 seconds. This allows us to report approximate results even for large BJSS instances. The results obtained here outperform the best results in the literature which validates our idea of using the B&B as an approximate method. The majority of the today's supercomputers are occupied with both multi-core CPUs and GPUs. Our previous approaches exploit one computing architecture at a time which may result in the underutilization of either the CPU or the GPU resources; thereby, wasting a significant computing power. For this reason, we propose in the fourth section of this chapter hybrid approaches that combine both parallel tree exploration and node-based parallelization exploiting both the multi-core CPU and GPU simultaneously. This hybridization is based on the concurrent kernels' execution provided by Nvidia Multi Processes Service (MPS) i.e. Multiple host processes (master and workers) can execute simultaneously their kernels on the GPU to accelerate the bounding of one or several nodes at a time. In order to ensure the validity and the relevance of the proposed approaches, a detailed experimentation section is given in the fifth section of this chapter. Using our proposed approaches, we have been able to increase the relative speedup to reach 160 times faster compared to our optimized serial B&B version. Moreover, we have been able to solve optimally ten benchmark instances never solved before.

Solving this problem by the means of the B&B algorithm is inefficient for dealing with large benchmark instances due to their huge search space. The only choice for dealing with such instances is to use approximate methods. These methods perform a trade-off between the quality of the obtained solution and the explored search space, hence, the running time. Chapter 5 is dedicated to the approximate resolution of the BJSS problem using the Tabu Search (TS) metaheuristic. Thanks to its high adaptability, it represents one of the most widely used metaheuristics for solving complex scheduling problems. This method represents a higher level heuristic procedure designed to guide other heuristics to escape the trap of local optimality. The first section of this chapter focuses on proposing an efficient adaptation of the TS algorithm for the BJSS problem. Indeed, the previously proposed metaheuristics for the BJSS problem had low results quality in terms of Makespan. This is due essentially to the fact that the classical reproduction techniques do not take into account the specification of the blocking constraint, resulting in a massive exploration of unfeasible solutions. The same problem is encountered with TS method. i.e., In most cases, the classical neighborhood

function produces infeasible solutions, leading to a useless exploration of the search space and a poor obtained solution quality. To overcome this drawback, we first propose in this chapter an efficient sequential TS algorithm adapted to the blocking constraint by proposing a new neighborhood function based on the reconstruction strategy. Our proposed neighborhood consists to remove arcs causing the infeasibility and rebuild the neighboring solutions by using heuristics. Our goal from this neighborhood is to guide the search process to explore only feasible solutions using heuristics that allow avoiding a random exploration of the search space. The second section of this chapter describes our proposed parallelization schemes that exploit HPC architectures. As it turns out, using heuristics to recover the feasibility of the neighboring solutions is a time-consuming operation which makes the TS algorithm very slow, since it spends a huge time to explore a small area in the search space. To overcome this drawback, we additionally propose several parallelization approaches of our proposed serial TS implementation. The parallelization allows on one hand to accelerate the search process of the TS method, and on the other hand to diversify it. The TS algorithm is very well adapted for parallelization, giving us the possibility of exploring simultaneously several areas in the search space. The first parallel approach which exploits the multi-core CPUs of a single machine, aims to accelerate the TS running time without changing the algorithm structure or the search trajectory. Indeed, this approach doesn't aim to improve the solution quality since the search space is explored exactly in the same way as in the serial version. Instead of evaluating one neighbor solution at a time, this approach evaluates all neighboring solutions at ones using multiple threads exploiting the multi-core CPU processors. The goal of our second parallel approaches is to improve the solution quality,*i.e.,* reporting the impact of parallelism on the obtained solution quality. These approaches exploit a cluster-based supercomputer where a large number of parallel processes explore the search space simultaneously using one or several search strategies. In order to show the quality of the proposed neighborhood and the performance of our parallelization approaches, a detailed experiments and discussions are given in the third section of this chapter. The obtained results indicate that our proposed serial and parallel TS approaches outperform almost all the stat of the art results.

Finally, the Conclusion of this thesis will focus on the general appreciation of the results obtained by the parallelization and the identification of future perspectives of our work.

2

Scheduling Problem and Blocking Job Shop Scheduling Problem

2.1 Introduction

The goal of our thesis is to solve the Blocking Job Shop Scheduling Problem using High-Performance Computing (HPC) Architectures. Before presenting our solutions, we introduce the reader to some basic notions on scheduling problems. Indeed, the first part of this chapter is dedicated to general information about scheduling problems and their classifications. Two classifications exist: The first one is based on the $\alpha|\beta|\gamma$ notation used to describe formally a scheduling problem. The second one uses complexity theory to classify scheduling problems according to the time and memory space used to solve them. This latter classification is based on the deducing the complexity of a problem by reducing it to another problem whose is known in advance. After that, we introduce the most widely used resolution methods to solve scheduling problems. The second part of the chapter is dedicated to the description of the Blocking Job Shop Scheduling problem (BJSS) and its mathematical formulation, in

addition to a brief list of its application areas. After that, we will give a detailed description of the modelization used to solve this problem.

The last part of the chapter contains some important application areas of the BJSS problem that have a high economic impact. Indeed, due to the importance of the blocking constraint and its economic impact, the BJSS problem has been extensively studied. However, most of the existing approaches are sequential and don't take into account the specificity of the blocking constraint. The aim of this thesis is to fill these important gaps by proposing parallel methods that take into account the blocking constraint.

2.2 Generality on Scheduling Problems

Scheduling can be encountered in various real-world problems. Due to the huge number of existing scheduling problems, the research community has defined a standard notation in which a definition of a scheduling problem can be formally expressed. In the following, we describe this notation after introducing the basic concepts that are common to all the scheduling problems.

2.2.1 Basic Concepts

Several definitions of a scheduling problem exist in the literature. According to Carlier and Chretienne [12] "Scheduling consists in forecasting the processing of a work by assigning resources to tasks and fixing their start times. [...] The different components of a scheduling problem are the tasks, the constraints, the resources and the objective function. [...] The tasks must be programmed to optimize a specific objective [...]. However, it will often be more realistic to consider several criteria." In other words, a scheduling problem can be seen as the assignment of **resources** to the **jobs** while optimizing some **criteria** and satisfying certain **restrictions**. Hence, defining a scheduling problem consists to define its basic components: jobs, resources, objectives, and constraints.

Jobs:

A job (Product) is a sequence of operations, it needs the utilization of a set of resources in order to be finalized. An operation represents an elementary work that consumes a certain number of units from a given resource. An operation can be characterized by a starting time

t_i, a finishing time c_i, and a processing time p_i. In addition, we also have the release time r_i of an operation which represents the earliest starting time of an operation and the due date d_i which represents the latest ending time of an operation. An operation is called *preemptive* if its execution can be divided into several pieces. A problem containing *preemptive* operations is called a *preemptive* problem; Otherwise, the problem is *non-preemptive* [70].

Resources:

A resource is a physical or human mean intended to be used by a given operation and it is available in limited quantity. The availability of a resource k is characterized by the parameter h_k, and may vary in time according to a function $h_k(t)$. Two categories of resources can be distinguished, namely, *renewable* and *non-renewable* resources. A resource is said to be *renewable* if after being used (allocated) by an operation it becomes available again with the same quantity for the subsequent operations. In other words, the available quantity is renewed from one operation to another in such a way that it remains constant. There are two types of renewable resources: *disjunctive* and *cumulative*. A resource is called *disjunctive* if it can only handle one operation at a time; Otherwise, a resource is called Cumulative. i.e. It can handle several operations simultaneously. Similarly, a resource is called *non-renewable* if after being used by an operation, it is no longer available for other operations in the same quantity. Hence, its consumption over time is limited.

Restrictions:

The constraints represent the limits imposed by the environment. They express restrictions on values that can be taken by one or more variables in the definition of a scheduling problem. Several kinds of constraints exist in the literature, among which we can find the following: temporal constraints such as due date and release time; precedence constraints which express order or synchronization between operations; and resource constraints such as availability constraints and resources utilization constraints [7].

Goals:

The goal of solving a scheduling problem is to optimize one or several objective functions. Several kinds of objectives exist in the literature, among which we can mention the following: temporal objectives such as minimizing the total completion time or the average completion

time;resources related objectives such as maximizing the resources load or minimizing the number of resources needed to complete a set of operations; and cost related objectives which ask to minimize the costs of the production, storage, and transportation.

2.2.2 Classification of Scheduling Problems

In the literature, the theory of scheduling includes a huge number of scheduling problems with different complexities. In order to simplify their identification Graham *et al.* [41] proposed the $\alpha|\beta|\gamma$ notation.The parameter α describes the environment of the machines, the parameter β specifies the characteristics of the jobs, and γ represents the optimality criteria.

Job Characteristics:

The job characteristics are specified by the parameter β. This parameter represents the concatenation of six values $\beta 1$, $\beta 2$, $\beta 3$, $\beta 4$, $\beta 5$, and $\beta 6$.

$\beta 1$ denotes whether preemption is allowed or not. In case of allowed preemption, $\beta 1$ appears in β and takes the value pmtn ($\beta 1$ = pmtn); otherwise, $\beta 1$ does not appear in β. As seen earlier, preemption consists to interrupt the processing of a job or an operation which can be resumed later on the same or on another machine without additional costs; this action can be repeated several times.

$\beta 2$ denotes the precedence relations between jobs which can be represented by an acyclic directed graph. A precedence relation between two jobs expresses the fact that one job must be completed before the other one starts. If precedence relations are allowed, $\beta 2$ appears in β and takes the value $\beta 2$ = prec. Otherwise, if there are no precedence constraints, $\beta 2$ does not appear in β. Several other types of precedence relations exist in the literature. For more information, the readers may refer to [41].

$\beta 3$ describes the release dates of jobs. In case of $\beta 3 = r_i$, the release dates must be specified for each job. If $r_i = 0$ for all jobs, then $\beta 3$ does not appear in β.

$\beta 4$ denotes restrictions on the processing times or on the number of operations. If $\beta 4$ takes the value $p_i = p$, then all operations have the same running time which is p units; otherwise, $\beta 4$ does not appear in β.

$\beta 5$ denotes deadlines for jobs. If $\beta 5 = d_i$, then a due date d_i is specified for each job J_i, which means that the job must be finished before time d_i. Otherwise, this parameter does not appear in β.

In some scheduling applications, jobs can be grouped into batches. If that is the case, then no deadline is specified.

Finally, $\beta 6$ denotes the batching problems. A batch is a set of jobs (one to several jobs) which must be processed together on a specified machine. Two kinds of batching problems exist in the literature namely, p-batch problems and s-batch problems. In a p-batching problem, the length of a batch is equal to the maximum processing times of all jobs in the batch, while in s-batch problems, the length of a batch is equal to the sum of the processing time of all jobs in the batch. If $\beta 6$ is equal to p-batch or s-batch, then $\beta 6$ appears in β; otherwise, It doesn't appear in β.

Machine Environment α:

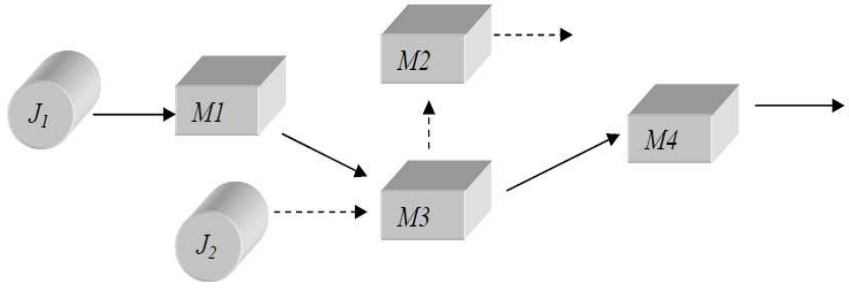
The machine environment is characterized by the concatenation of two parameters $\alpha 1$ and $\alpha 2$, i.e. $\alpha = \alpha 1 \alpha 2$. The parameter $\alpha 1$ can take the values *, P, Q, R, PMPM, QMPM, G, X, O, J, and F.

If $\alpha 1$ belongs to the set *, P, Q, R, PMPM, QMPM , then all jobs consist of a single operation. Moreover, if $\alpha 1 = *$, each job must be executed on a specified machine. If $\alpha 1$ belongs to the set {P, Q, R} then, parallel machines are considered, i.e., each job can be processed by several machines. P denotes identical parallel machines where the processing time of a job is the same on all machines. Q denotes uniform parallel machines where each machine has its own speed. In this case, the processing time of a job differs from one machine to another one. Finally, R denotes unrelated parallel machines where there are job-dependent speeds. $\alpha 1$ = PMPM denotes multi-purpose machines with identical speeds, whereas $\alpha 1$ = QMPM denotes multi-purpose machines with uniform speeds.

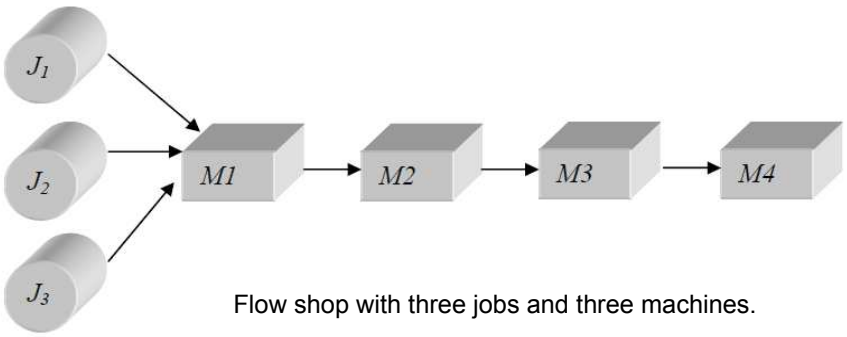
If $\alpha 1$ belongs to the set {G, X, O, J, F}, then, a multi-operation model is considered, i.e., each job has a set of operations.

If $\alpha 1$ = G, then a general shop model is considered. Job shops, flow shops, open shops, and mixed shops are special cases of the general shop. In a job shop model, indicated by $\alpha 1$ = J, we have special precedence relations between the job's operations. In other words, each job has its own way of crossing on machines. Furthermore, Figure 2.2.1 shows an example of job shop problem with two jobs and three machines.In addition, a job shop is said to be with machine repetition if a job can pass on the same machine several times.

The flow shop is indicated by $\alpha 1$ = F. It represents a special case of the job shop problem,



Job shop with two jobs and three machines.



Flow shop with three jobs and three machines.

Figure 2.2.1: Shop scheduling problem.

in which all jobs have the same way of crossing on machines as shown in Figure 2.2.1. The open shop, denoted by $\alpha 1 = \mathrm{O}$, can be described as a flow shop problem with no precedence relations between the job's operations. Finally, a mixed shop, indicated by $\alpha 1 = \mathrm{X}$, is a combination of a job shop and an open shop problems.

Optimality Criteria:

The optimization criteria are expressed as a function called *objective function* which represents the goal that we are trying to reach by solving a scheduling problem. The objective function can be a minimization function or a maximization function of one or several criteria. In other words, the goal of a scheduling problem is to find a feasible schedule which minimizes or maximizes an objective function. These criteria are represented by the field γ.

The most widely used objective functions are the ones based on finishing times of jobs. The most popular one is the *makespan* denoted by $\gamma = Cmax$ and which is equal to $max\{C_i/i =$

$1, ..., n\}$, where C_i represents the finishing time of job J_i.

Other objective functions exist in the literature. These functions depend on the due date d_i associated to the job J_i $/i = \{1, ..., n\}$, where n represents the number of jobs. Among these criteria we can find the following:

- *Lateness* : $Lmax = Max\{L_i\}$ *where* $L_i = C_i - d_i, /i = \{1, ..., n\}$
- *Flowtime* : $Fmax = Max\{F_i\}$ *where* $F_i = C_i - r_i, /i = \{1, ..., n\}$
- *Tardiness* : $Tmax = Max\{T_i\}$ *where* $T_i = Max\{L_i, 0\}, /i = \{1, ..., n\}$
- *Earliness* : $Emax = Max\{E_i\}$ *where* $E_i = Max\{-L_i, 0\}, /i = \{1, ..., n\}$
- *absolute deviation* : $Emax = Max\{D_i\}$ *where* $D_i = |C_i - d_i|, /i = \{1, ..., n\}$
- *squared deviation* : $Smax = Max\{S_i\}$ *where* $S_i = (C_i - d_i)^2, /i = \{1, ..., n\}$

There are also other criteria that correspond to the arithmetic mean or the weighted sum of the criteria already presented [10, 41, 70].

An objective function which is non-decreasing with respect to all variables C_i is called regular. Functions involving E_i, D_i, S_i are not regular[10].

A schedule is said to be *active* if there is no possibility to schedule jobs (operations) earlier without violating some constraint. A schedule is said to be *semi-active* if no job (operation) can be processed earlier without changing the processing order or violating the constraints.

2.2.3 Examples of $\alpha|\beta|\gamma$ notation

To illustrate the $\alpha|\beta|\gamma$ notation, we will present some examples.

The first example of this notation $P|\ prec;\ p_i = 1 \mid Cmax$ represents a scheduling problem that contains n jobs with single-unit processing time, precedence constraints between jobs on identical parallel machines, and Makespan (Cmax) as an optimization criterion.

The second example $J3 \mid p_i = 1 \mid Cmax$ represents the problem of minimizing the *makespan* (Cmax) for a job shop scheduling problem with three machines, in which the processing time of all operations is equal to one single unit.

2.3 Complexity Theory

In this section, we give a short introduction to the the theory of computational complexity. For that purpose, we will need to define the concepts of decision problem and introduce some classes of complexity. But before that, we present some basic definitions.

2.3.1 Algorithm Complexity

The amount of computational resources used by an algorithm is essential to judge its efficiency. Generally, two resources are considered as important: the time taken by the machine to execute the algorithm and the amount of space used by the algorithm. The resources (time and space) used by an algorithm are generally evaluated using an asymptotic notation, called O-notation. In this notation, we ignore all constants and all terms that tend to zero when divided by another term, and when the size of the problem tends to infinity. The time complexity of an algorithm is the number of necessary instructions (assignment, comparison, algebraic operations, read and write, etc.) that the algorithm performs during its processing. For discrete problems, this number can be represented by a function "g" that depends on the number of inputs and outputs of the problem and on the magnitude of the largest input.

As an example, we consider an algorithm whose maximal number of executed instructions is $g(n) = cn^2 + n$ where n represents the size of the problem instance (the number of inputs and outputs) and c is a constant. In this case, we say that the algorithm has a complexity of $O(g(n))$, where $g(n)$ can be simplified to $g(n) = n^2$.

We say that an algorithm has a *polynomial complexity* if its number of executed instructions ($g(n)$) is upper bounded by a polynomial function that depends on the input size n. If this function depends on the size of the data and the magnitude of it biggest element, then the algorithm has a *pseudo-polynomial complexity*. In other cases, we say that the algorithm has an *exponential complexity*.

The complexity of an algorithm can be computed either by counting the number of iterations or by splitting the algorithm into sub-algorithms of known complexity. The complexity in the latter case can be the multiplication or the addition (or combination of the additions and multiplications) of the complexities of its sub-algorithms according to the program structure.

2.3.2 Complexity of Scheduling Problems

The experience shows that some computational problems are easier to solve than others. Complexity theory provides a mathematical framework in which computational problems are studied so that they can be classified as "easy" or "hard" [10]. In the following, we present some of the important classes of complexity. For more detailed presentation, the readers may refer to [33], [9], and [49].

Due to the difficulty of classifying problems directly, we associate for each studied problem a decision problem. A decision problem is a problem whose solution is either YES or NO. Hence, we don't classify the problem itself, but its associated decision problem.

In the following, we will introduce some important notions in the theory of complexity. Our starting point will be to define decision problems. A decision problem for an optimization problem, in which the goal is to minimize an objective function, can be formulated as a simple question in the form "Is there a solution for which the objective function has a value below z?". If a decision problem of an optimization problem admits a polynomial algorithm, then, a polynomial algorithm that solves the associated optimization problem will also exist. Similarly, if a polynomial algorithm is known for an optimization problem, then there is also a polynomial algorithm for its associated decision problem. This property allows us to categorize problems through their decision problems. Thus, in what follows we will only consider the classification of decision problems [10].

P and NP classes

The P class, where P stands for Polynomial, contains all the so-called easy problems. More precisely, the class P contains all decision problems for which a polynomial algorithm that allows to solve them exits. There are problems for which no polynomial algorithm exists. The class NP (where NP stands for non-deterministic polynomial) contains many of those difficult problems. Therefore, the NP class contains more difficult problems compared to the P class. The class NP contains all decision problems that can be solved in non-deterministically in polynomial time. Equivalently, these problems are the ones for which a solution can be checked in polynomial time. Each problem in P is also included in NP (that is $P \subseteq NP$). The question of whether $P = NP$ is the central problem of computation complexity theory, and is probably the most important problem of computer science. The class of NP-complete problems contains the most difficult problems in the class NP.

NP-complete Class Problems

The classification of problems as NP-complete relies on the concept of reduction. More precisely a problem A is classified as NP-complete if it is in NP, and there exists a polynomial time reduction from any NP-complete problem to problem A.

Concept of Reduction:

The reduction converts one problem into another so as to allow to use the solution of the second problem to solve the initial one. To help the reader to understand this concept, we will take a very well-known real-life example. Let's say that you don't know the city, and you have to go to a specific location. In order to seek your way, you will first look for a map of the city. Hence, the problem of finding your way has therefore been reduced to the problem of finding a map of the city.

In case of optimization problems, a problem is often a special case, equivalent to, or a generalization of another problem. Reduction allows to reduce the complexity of solving a problem by transforming it into another problem whose complexity is already known.

Let's define A and B as two decision problems. The problem A is the one to be solved and B is the problem whose complexity is known to us. A is reducible to B (denoted by A μ B) if and only if there is a function f of polynomial time complexity that transforms the instances of A into those of B, so that the answer for A is yes if and only if the answer for B is yes [67]. Reduction is a transitive relation, that is, if A is reducible to B (A μ B) and B is reducible to C (Bμ C), then A is reducible to C (A μ C). Polynomial time reductions can be use in two different ways to infer the computational complexity of problems: if A μ B and if there is a polynomial algorithm for B then, there will be also a polynomial time algorithm for A. Conversely, if there is no polynomial algorithm for A then, there is no polynomial time algorithm for B. As mentioned earlier, a decision problem is said to be NP-complete if any problem of the class NP can be reduced polynomially to it. By using the reduction concept, we can say that if a decision problem B is known to be NP-complete and a decision problem A such that B μ A then, A is NP-complete. By using these proprieties, Cook, in 1971, has proved the existence of the NP-completeness class by showing that any problem in the class NP can be reduced to the SAT problem[18], which means that the SAT problem is in some sense the hardest problem in the class NP. Then to prove that a problem P is NP-complete, it suffices to find a polynomial time reduction from SAT to P, either directly or indirectly (using the transitivity property).

The definition of the NP-complete class is very strong. If we can find a polynomial time algorithm for a single NP-complete problem, then by the concept of reduction, all NP-complete problems will be solvable in polynomial time. Figure 2.3.1 shows the different complexity classes discussed in this section.

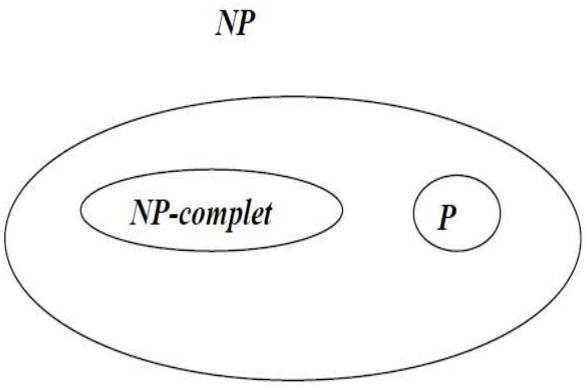


Figure 2.3.1: Complexity classes.

The theory of complexity is interested in the study of problems classification and the existing boundaries between the different classes. It is clear that the class P is included in NP, but the problem of whether 'P = NP' is still widely open. The expectation of most complexity theorists is that this equality is false, but up to now, the progress toward solving that question has been very slow. Some authors make a distinction between NP-hard and NP-complete notations. Indeed, an optimization problem is called NP-hard if its associated decision problem is NP-complete. We say that an optimization problem is NP-hard in the strong sense if there is no pseudo-polynomial algorithm that allows to solve it unless P = NP. Similarly, we say that an optimization problem is weakly NP-hard if there is a pseudo-

polynomial algorithm that allows to solve it.

The classes presented in this chapter are the two most important classes of the complexity theory. However, many more classes have been defined and studied. For more details, the reader may refer to [18, 67, 70].

2.4 Solving Scheduling Problems (Resolution Methods)

In this section, we focus on describing the main resolution methods used to solve optimization problems. There are two main categories: exact and approximate methods.

2.4.1 Exact Methods

The exact methods consist to return the optimal solution along with a proof of optimality. Among these methods we find:

Branch and Bound Algorithms

Branch and Bound (B&B) algorithms were introduced by A. H. Land and A. G. Doig in 1960 [51]. They are based on the implicit and intelligent enumeration of all feasible solutions. These algorithms are based on three operators: Branching, Bounding, and pruning. Branching is a recursive process that consists in dividing the initial problem into several smaller sub-problems, where each one is treated recursively in the same way. Therefore, we solve the initial problem (reporting the best-explored solution) by solving its sub-problems. This process can be represented by a search tree that allows enumerating all feasible solutions leading to improve the current best solution. The bounding process evaluates the ability of a given sub-problem to improve the current best solution. The pruning process uses the bounding results to eliminate the unpromising branches that can not lead to improve the current best solution. In addition, we can mention the exploration strategies which represent the way the search tree is explored; the most known exploration strategies are Breadth-first, Depth-first, and Best-first.

Cutting-Plane Method

The Cutting-Plane method, developed by Schrijver in 1986 [66], is intended to solve combinatorial optimization problems which are formulated in the form of a linear program. The

method consists to refine iteratively a set of feasible solutions by means of linear inequalities. This method is known to be inefficient for large NP-hard optimization problems.

Branch and Cut Method

The Branch and Cut method combines the Branch and Bound algorithm with the Cutting-plane method in order to deal with large problems. The method begins by relaxing the problem using the branching process and then applies cutting-plane method. If no solution is found by the Cutting-Plane method, the problem will be splitted again into several sub-problems which are treated in the same way.

2.4.2 Approximate Methods

The exact resolution methods allow to report the optimal solution for optimization problems. However, the time needed for these approaches is huge. The approximate algorithms perform a tradeoff between the time complexity and the quality of the obtained solution by reporting a near optimal solution in reasonable time. Two categories of approximate algorithms exist in the literature: Heuristics and Metaheuristics.

Heuristics

A heuristic [62] is an approximate algorithm designed to solve an NP-hard combinatorial optimization problem in a polynomial time by providing a feasible solution, usually of low quality. In most cases, a heuristic is designed for a particular problem and can not be applied for other problems. This fact has lead researchers to design alternative resolution methods that can be applied to different kinds of problems.

Metaheuristics

Metaheuristics are mathematical methods that can provide good solution quality for various optimization problems. Metaheuristics can be classified into two main categories. We can distinguish those who work with a population of solutions (population-based metaheuristics) from those who only handle one solution at a time (local search metaheuristics). The local search metaheuristics build a trajectory in the search space while trying to move towards the optimal solution. The best-known examples of these methods are Tabu Search and Simulated Annealing metaheuristics. Population-based metaheuristics are often inspired by

natural systems. The most well-known examples of these methods are Genetic Algorithms, Particle Swarm Optimization, and Ants Colony Algorithms.

To illustrate local search and evolutionary metaheuristics, we will present in the following one important example from each family, Tabu search for the first one, and genetic algorithms for the second one .

Tabu Search:

Tabu Search (TS) is a local search metaheuristic that uses a set of techniques that allow to avoid the trap of a local minimum and the repetition of explored solutions. This method which was introduced mainly by Glover in 1986 [37, 38], has shown great efficiency in solving NP-hard optimization problems. This method explores the search space by moving step by step from one solution to another towards the optimal one. The initial solution x from which the method starts is randomly generated. The transition from the solution x to another solution x' is obtained by applying a neighborhood function that consists to perform basic modifications on the solution x to obtain a set of neighboring solutions denoted by $N(x)$, and x' represents the best solution in this set. The algorithm uses a Tabu List (TL) of length k containing the k last visited solutions. The goal of TL is to avoid exploring the same sequence of solutions in the short term. To that effect, the choice of the next solution (x') is made by taking into account the elements of the TL. Finally, the exploration process is interrupted when one or more stopping criteria are satisfied. Moreover, The general exploration process of the TS method is illustrated in Figure 2.4.1.

Genetic Algorithms:

Genetic algorithms (GA) are optimization methods inspired by the mechanism of genetics. They have been adapted for optimization problems by John Holland in 1975 [39] and subsequently improved by David Goldberg in 1989 [40]. Unlike local search methods that involve a single solution, genetic algorithms manipulate a group of solutions at each stage of the search process. The main idea of the algorithm is to use the collective properties of a set of solutions, called population, in order to effectively guide the search towards good solutions in the search space. Genetic algorithms have a very simple mechanism. The algorithm starts from a randomly generated population of potential solutions, called chromosomes. The solutions are then evaluated using a relative fitness function. Based on the evaluation of each chromosome, a new population is created by using simple evolutionary operations (selection,

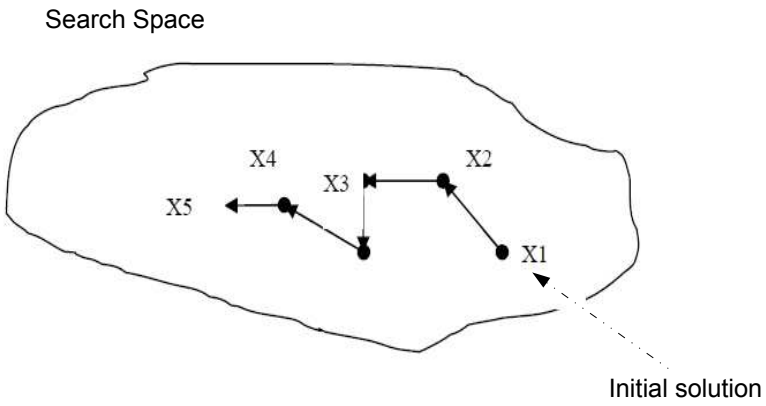


Figure 2.4.1: Tabu Search exploration.

cross-over, and mutation); in which, some individuals reproduce, others disappear and only the best-adapted individuals are expected to survive. This process is repeated until stopping criteria are reached. The most important aspect when adapting the genetic algorithm is the codification that characterizes each chromosome in the population. The implementation of a genetic algorithm requires three elements: The selection, the cross-over, and the mutation. The selection process allows to identify the best individuals in a population and to eliminate the bad ones while passing from one generation to another. This process is based on the fitness function related to each individual. The cross-over process allows the exploration of the search space and ensures the diversity of the population by manipulating the structure of the chromosomes. The cross-over generates two children by using two parents, hoping that at least one of the two children will inherit good genes from both parents. The mutation process performs a small change in the genetic code of chromosomes. The goal here is to diversify the population and thus avoid falling into local optimums. This operator is applied

with a low probability as compared to the cross-over operator.

2.5 Job Shop Scheduling Problem With The Blocking Constraint

2.5.1 Problem Formulation

The classical JSSP can be defined by a set J of n jobs $(J1, ..., Jn)$ to be processed on a set M of m machines $(M1, ..., Mm)$. Each machine can process at most one job at a given time. The execution of a job on a machine is called operation. We denote by O the set of all operations $(o_1, ..., o_{n*m})$. Each operation o_i needs to use a machine $M(i)$ for an uninterrupted duration called processing time p_i. Each job has its own sequence of crossing on machines which creates precedence constraints between consecutive operations of the same job. A solution (schedule) for this problem consists to assign a starting and finishing times t_i and c_i for each operation o_i $(i = 1, ..., n * m)$ while satisfying all constraints. Our goal is to minimize the *Makespan* $(Cmax)$.

In order to describe the problem in a formal way, we use a mathematical formulation of the classical job shop scheduling problem which is often used in the literature. Let's define P_c as the set of all precedence constraints, and U_k the set of concurrent operations on machine k.

$$t_i + p_i \leq t_j \ \forall (o_i, o_j) \in P_c. \tag{2.1}$$

$$(t_j - t_i) \geq p_i \ or \ (t_i - t_j) \geq p_j \forall (o_i, o_j) \in U_k / k \in \{1, , m\}. \tag{2.2}$$

$$t_i \geq 0 \quad \forall o_i \in O. \tag{2.3}$$

Where t_i and p_i represent respectively the starting and processing times of operation i (o_i).

In this formulation, only one of the two inequalities of 2.2 must be valid. Equation 2.1 describes the precedence order of the operations of the same job. Equation 2.2 describes the execution order of concurrent operations on the same machine k.

The goal considered here is the minimization the Makespan which represents the finishing time of a schedule.

2.5.2 Constraints

In addition to the precedence constraint already presented, the JSSP problem contains several constraints that involve both the possibilities of using the machines and the links that may exist between operations. Among these constraints:

- The machines are independent of each other.
- The jobs are independent of each other, i.e., there is no order or priority attached to jobs.
- A job can only be processed by one machine at a time.
- A machine can only process one operation at a time.
- Only the effective execution time is taken into account, i.e., the transportation time from one machine to another, preparation, etc. are not taken into consideration.
- The machines are available until the end of the scheduling. In particular, machine failures are not taken into account.

Other constraints which are not directly connected to the JSSP problem may exist; among them:

- A running operation cannot be interrupted (no preemption is allowed.).
- Operations are allowed to wait for resources as much as needed (no due dates).
- The storage capacity between machines is considered as unlimited.

The Job Shop Scheduling with the already presented constraints is called *Classical Job Shop Scheduling Problem*. In this thesis, we consider a job shop scheduling with no-storage space between machines which induces a blocking situation. This later problem is called the *Blocking Job Shop Scheduling* (BJSS).

Blocking Constraint:

The classical job shop scheduling problem (JSSP) assumes an unlimited intermediate buffer capacity between consecutive operations of a job which is impossible in real-world problems including, the ones that arise in manufacturing. The BJSS problem is a version of the classical JSSP with no intermediate buffers between machines. Hence, a job has to wait on the current machine until the next one becomes available for processing. In other words, a job which has completed its processing time on a machine stays in it (blocks it from processing other jobs) until its next machine become available for processing.

By adding the blocking constraint, the mathematical formulation of the problem will be as follows: we denote by $o_{j\prime}$ the operation that immediately follows operation o_j in the same job and $t_{j\prime}$ its starting time. In the case where $o_{j\prime}$ doesn't exist (o_j is the last operation in the job), we set $t_{j\prime} = t_j + p_j$.

$$t_i + p_i \leq t_j \ \forall (o_i, o_j) \in P_c. \tag{2.4}$$

$$(t_j - t_i) \geq p_i \ or \ (t_i - t_j) \geq p_j \ \ \forall (o_i, o_j) \in U_k / k \in \{1, , m\}. \tag{2.5}$$

$$t_i \geq 0 \ \ \forall o_i \in O. \tag{2.6}$$

$$(t_j \geq t_{i\prime}) \ or \ (t_i \geq t_{j\prime}) \ \ \forall (o_i, o_j) \in U_k / k \in \{1, , m\}. \tag{2.7}$$

As compared to the mathematical formulation of the classical JSSP, we added a new equation 2.7. This latter expresses the fact that one operation must leave the machine before beginning the processing of the other one. For example, the first inequality ($t_j \geq t_{i\prime}$) indicates that operation o_j can not start until operation o_i leaves the machine which is the same time that operation $o_{i\prime}$ starts.

There are two different cases of the BJSS depending on the application area and the specification of the manufacturing system, namely, the blocking with swap allowed and the blocking with no-swap cases. The swap situation is specific to the blocking constraint. It represents a deadlock situation between two or more operations with zero-length cycle in the graph representation. Figure 2.5.1 shows a swap situation between three operations, where

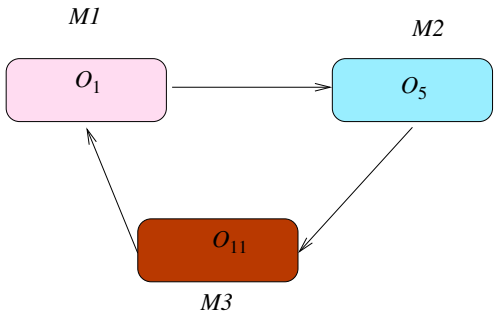


Figure 2.5.1: Swap situation between three operations.

each one is waiting for the liberation of a machine occupied by another operation in the same swap.

The solution of the swap situation in the blocking with swap allowed, consists to move simultaneously the swap operations to their subsequent machines as shown in Figure 2.5.2. We should mention that all swap operations have the same processing time which is equal to the maximum processing time of the operations in the swap.

In the blocking no-swap case, a solution that contains a swap situation is considered as infeasible.

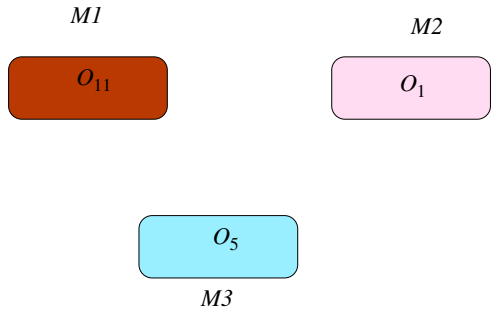


Figure 2.5.2: Solution of the swap situation (Figure 2.5.1) in the BWS problem.

2.5.3 Modelization of the BJSS problem

The *disjunctive graph model* was the first modelization proposed to present the classical JSSP. It was introduced by Roy and Sussman [65] in 1964. Based on this modelization, Mascis et al. [54] proposed the *alternative graph model* to present the BJSS problem. This model can be defined as a graph $G = (N, F, A)$ where N represents a set of nodes (operations)

with two additional dummy nodes (start and finish) modeling the start and the finishing of the schedule. F represents a set of fixed arcs imposed by precedence constraints between consecutive operations of the same job and f_{qp} is the length of arc $(q,p) \in F$. Finally, A is a set of alternative pairs $((i,j),(h,k))$ representing the processing order for concurrent operations on the same machine and a_{ij} is the length of alternative arc (i,j). Each arc represents the fact that one operation must be completed before starting the processing of the other operation. A selection S_1 is a set of arcs obtained from A by choosing at most one arc from each alternative pair. We call $G(S_1) = (N, F \cup S_1)$ the alternative graph representation of the selection S_1. We say that a selection S_1 is feasible if there is no positive length cycle in $G(S_1)$. The evaluation (*Makespan*) of S_1 is the longest path in $G(S_1)$. We say that S_1 is a complete selection if exactly one arc is chosen from each pair, therefore $|A| = |S_1|$. We define a schedule (solution of the problem) as a complete feasible selection. Finally, given a feasible selection S_1, let $l(i,j)$ be the length of the longest path from operation i to j in $G(S_1)$.

We call the last operation of each job (example o_r) an *ideal operation* because the machine becomes immediately available after the end of the operation processing time (p_r). Otherwise, the operation (example o_i) is called a *blocking operation*, in this case, we denote by $\sigma(i)$ the operation immediately following o_i in the same job.

Table 2.5.1: BJSS instance with two jobs and three machines.

job	sequence	processing times
J_1	M_1, M_2, M_3	5, 3, 8
J_2	M_2, M_1, M_3	8, 2, 7

Table 2.5.1 represents a BJSS instance with two products (jobs) and three machines. The first product (J_1) has 5 minutes processing time on machine M_1, 3 minutes processing time on machine M_2, and 8 minutes processing time on machine M_3. The second product (J_2) has 8 minutes processing time on machine M_2, 2 minutes processing time on machine M_1, and 7 minutes processing time on machine M_3.

Figure 2.5.3 represents an alternative graph of the BJSS instance in Table 2.5.1. This graph has three alternative pairs, two between blocking operations and one between ideal operations. Both operations 2 and 4 need the same machine M_2. Since M_2 can not process both operations at the same time, we associate these two operations with an alternative pair.

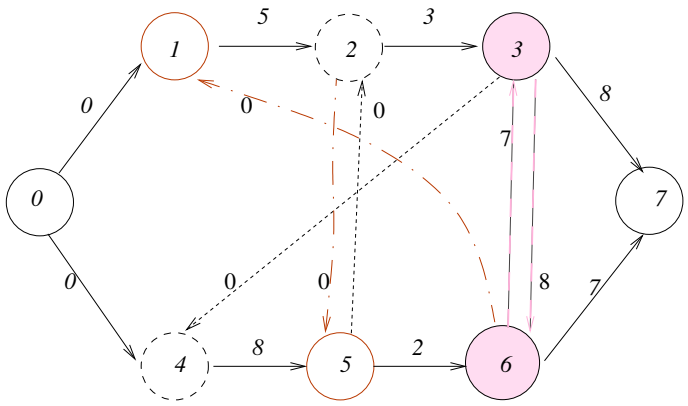


Figure 2.5.3: Graph representation of all alternative pairs for the BJSS instance of Table 2.5.1.

Since operations 2 and 4 are blocking operations the first alternative arc $(3, 4)$ represents the choice where operation 2 must be finished before the beginning of operation 4. His mate, arc $(2, 5)$ represents the choice whereby operation 4 must be finished before the beginning of operation 2. Similarly for the alternative pair $((2,5), (6,1))$ between operations 1 and 5. We use the same process to generate the alternative pair $((3, 6), (6, 3))$ with the exception that both operation 3 and operation 6 are ideal, i.e. the machine becomes immediately available after the end of the operations' processing time.

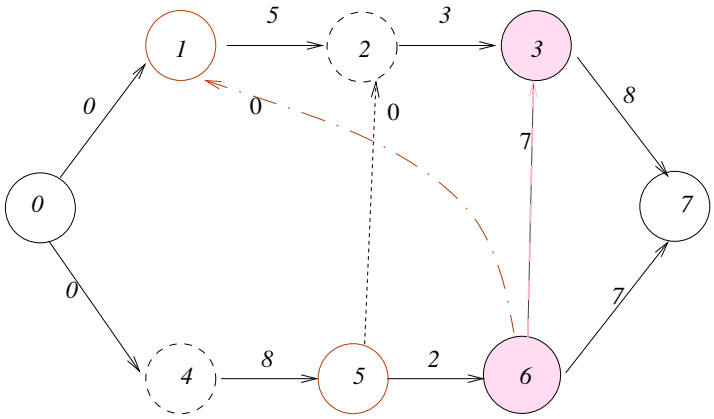


Figure 2.5.4: Schedule for BJSP in Table 2.5.1 whit $Cmax$=26.

Figure 2.5.4 represents a feasible schedule (solution) for the BJSS instance in Table 2.5.1, obtained by choosing one arc from each pair in the alternative graph of Figure 2.5.3. The *Makespan* of this schedule is equal to 26 ($Cmax = 26$) which is the longest path in the

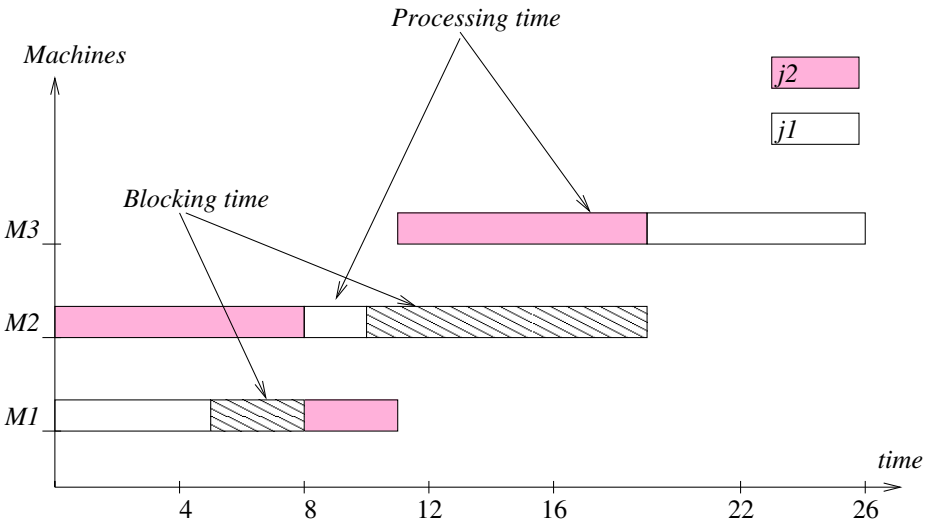


Figure 2.5.5: Gantt chart of the schedule in Figure 2.5.4.

obtained graph.

The Gantt chart in Figure 2.5.5 represents both the processing and blocking times of the solution of Figure 2.5.4. For example, after the end of its processing time the job J_1 blocks the machine M_1 until machine M_2 becomes available for processing J_1.

Forming the graph of all possibilities (Search graph)

The search graph represents the graph of all possibilities. It is obtained by generating all alternative pairs, knowing that each alternative pair represents the processing order between only two concurrent operations. Therefore, if a machine has four concurrent operations, we need six alternative pairs to show all the possibilities on how these operations are executed on the machine. In a general way if we have n concurrent operations on a machine, the number of alternative pairs is equal to:

$$Nb_{pairs} = \sum_{i=1}^{n} n - i$$

For a BJSS instance with ten jobs and ten machines, we have 45 alternative pairs for each machine. Since we have ten machines, the total number of pairs is equal to 450 alternative pairs. i.e. 2^{450} possibilities on how to choose the processing order.

In the following, we present how to generate the alternative pairs between every two concurrent operations.

Alternative Pairs Generation

Let us consider two blocking operations o_i, o_j and one ideal operation o_r, where $M(i) = M(j) = M(r)$. Since the three operations cannot be executed at the same time, we associate them with pairs of alternative arcs.

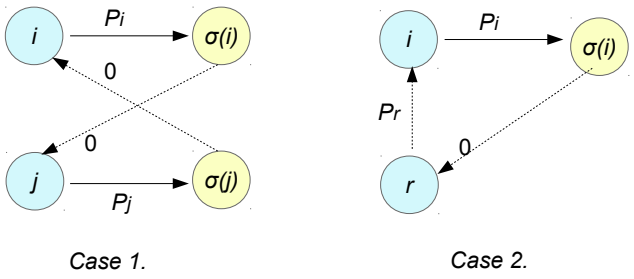


Figure 2.5.6: Alternative pairs between blocking and ideal operations.

Case 1) the alternative pair between operations o_i and o_j (Fig. 2.5.6): The first alternative arc $(\sigma(i), j)$ having length zero represents the situation where o_i is processed before o_j. Since o_i is a blocking operation, $M(i)$ cannot begin the processing of o_j until operation $o_{\sigma(i)}$ starts its processing on the machine $M(\sigma(i))$; which is the same time that o_i leaves the machine $M(i)$. The same process is followed for the other alternative arc$(\sigma(j), i)$.

Case2) the alternative pair between operations o_i and o_r (Fig. 2.5.6): It is the same process as in the first case for the alternative arc $(\sigma(i), r)$ since o_i is a blocking operation. Since operation o_r is an ideal operation, the machine $M(r)$ becomes immediately available after the end of its processing time. Thus, we add the alternative arc(r, i) with length p_r.

2.5.4 Blocking Job Shop Scheduling Complexity and Applications

The classical job shop scheduling problem is known in the literature to be NP-hard in the strong sense and its blocking extension (BJSS problem) appears to be even more difficult

to solve [10]. Indeed, Sotskov and shkhlevich [68] have proven that the job shop scheduling problem with three jobs and three machines $J3/n = 3/Cmax$ is NP-Hard. In addition, Lenstra and Rinnooy [53] have also proven that even the job shop problem with two machines, denoted by $J2//Cmax$, is NP-hard.

However, there are only a few special cases that can be solved in a polynomial and pseudo-polynomial time algorithms [45]:

- The flow shop scheduling problem represents a special case of the job shop scheduling problem in which all the jobs follow the same sequence of machines (for example: M1, M3, M4, M2). Gilmor et Gomory in 1964 [36] have proven that the flow shop problem with two machines and no-wait constraint ($F2/nwt/Cmax$) can be solved using a polynomial time algorithm in $O(n\ log\ n)$. However, the three machines version of it ($F3/nwt/Cmax$) is proven by Grey et al. [34] to be NP-hard. In the same way, the flow shop problem with two machines and blocking constraint ($F2/block/Cmax$) can be solved in a polynomial way since it is equivalent to the $F2/nwt/Cmax$ problem.

- The job shop scheduling problem with two machines and a maximum number of two operations per job, denoted as $J2/v_i \leq 2/Cmax$, can be solved in a polynomial time using the Jackson algorithm [47] since it can be easily transformed to the $F2//Cmax$ problem.

- The job shop scheduling problem with two machines and a fixed number of jobs denoted as $J2/n = k/Cmax$ is solved in a polynomial time by Brucker [10]. Similarly, the job shop scheduling problem with two machines and one unite processing time for all operations ($J2/P_{ij} = 1/Lmax$) is solved by Brucker with polynomial time algorithm in $O(r\ log\ r)$ [10].

  We can also mention that the job shop scheduling with m machines and two jobs denoted as $J/n = 2/cmax$ is solved in a polynomial time using a geometric approach proposed by Akers et al. in 1955 [3].

The blocking job shop scheduling problem (BJSS) appears to be even more difficult to solve as compared its classical version since an additional in-equation has been added to the mathematical formulation of the JSSP problem.

The number of all possible solutions for the BJSS problem is around $(n!)^m$. It should be noted that this number evolves faster according to the number of jobs (n) than according to the number of machines m. In other words, adding a job has a lot more impact on the complexity than adding a machine.

To give an idea about the search space, we take a small instance of ten jobs and ten machines. The number of possible solutions for this instance is around $3,9594 * 10^{63}$ which is huge. Enumerating all these solutions, using a serial computer, to get the optimal one is not feasible due to the huge running time. For this reason, exploiting high-performance computing architecture is crucial to accelerate this process.

The BJSS problem has several application areas among them, we can mention:

- the first interesting application of the BJSS problem is the trains scheduling. We have a set of trains, and each one has its journey from; for example, from terminus A on Algiers to terminus B on Oran. In order to reach its destination, Each train must pass through a set of rail segments, where each segment contains traffic lights to indicate its availability.

  To describe the trains scheduling as a BJSS problem, we model each train journey as a job and each rail segment as a machine. To avoid accidents, one train can use a segment at a given time. The blocking constraint appears in the case where a train reaches the end of a segment and the traffic light indicates that the next segment (machine) is used by another train. Thus, the train remains on the segment (blocks it from receiving other trains) until the traffic lights of the next one becomes green (the next segment is free) [19, 31].

- In a hospital, patients arrive on a random basis or by appointment for a consultation or hospitalization. A patient in our case represents a job, this patient needs a lot of resources: doctors, surgeons, consultation room, nurses, hospital operating room, post-operating room, etc. Some of these resources (e.g. operating room and post-operating room) cannot be shared with other patients and don't have waiting rooms; hence, inducing the blocking situation. For this reason it is necessary to schedule and monitor the patients flows in order to efficiently exploit these resources [27].

2.6 Conclusions

In this chapter, we introduced some of the basic definitions about scheduling problems and their classification using the $\alpha|\beta|\gamma$ notation. We also presented,in this chapter, another important classification which aims to classify problems according to their degree of difficulty. After that, we gave a quick overview on the most used methods that solve combinatorial

optimization problems. These methods can be grouped in two categories, exact and approximate methods: the exact methods aim to report the optimal solution of an optimization problem while the approximate methods aim to find a near-optimal solution in a reasonable time. The last part of this chapter was dedicated to the description of the problem treated in this thesis named, The Blocking Job Shop Scheduling (BJSS) Problem . This problem is known in the literature as NP-hard in the strong sense in which only a few special cases of it can be solved in a polynomial time. In the last section of this chapter, we presented An overview about some applications of the BJSS problem

In the next chapter, we describe the basic definitions related to High-Performance Computing Architectures.

3

High Performance Computing for Combinatorial Optimization

3.1 Introduction

The concept of parallelism has appeared at the same time as computer science itself, but only became accessible to other disciplines with the appearance of the first parallel machines. Indeed, with the availability of high-performance architectures at relatively low costs, many researchers tried to take advantage from these architectures to deal with complex problems that require huge computing time on serial computers. The primary goal of parallelism is to accelerate the time needed by a single processor to solve a complex problem using multiple processors concurrently. In order to achieve this goal, a general knowledge of the different parallelization concepts and basics information about parallel architectures and their classifications is therefore necessary. In the literature, several authors have made a detailed study on the parallelization of exact and approximate resolution methods. A presentation of some of these studies in addition to a more general literature review on parallel resolution

methods will be the subjects of this chapter.

3.2 High-Performance Computing Architectures and Parallel Algorithms

A sequential algorithm designed to solve a problem represents a set of instructions executed sequentially using a single processor. Akl [4] defines a parallel algorithm as a method for solving a problem using multiple compute resources simultaneously. Indeed, the problem is divided into several parts that can be solved simultaneously. To ensure the same result as the sequential algorithm, a control/coordination mechanism is used.

3.2.1 Computing Architectures

In the following, we introduce the basic information about high performing computing architectures and their classifications.

The von Neumann machine represents the most used example of sequential computers. It has four main components: the Central Processing Unit (CPU), the Control Unit, the Central Memory, and finally the Input/Output (I/O) devices [17]. The memory contains the sequence of instructions, representing the algorithm, that need to be executed on the CPU, in addition to the data that will be used during the execution. For each instruction in the algorithm, the control unit sends the instruction and the data related to it to the CPU. This latter uses registers to interpret and execute most of the computer commands.

The vast majority of today's stand-alone computers are parallel from a hardware perspective, since they contain multiple processing units (cores), each of which can execute multiple threads and/or multiple instructions in parallel.

The majority of the world's large supercomputers are clusters of stand-alone hardware connected together with a high bandwidth, low-latency network as an example we can cite *Infiniband* technology which is a computer-networking communications standard used in the high-performance computing architectures. For a recent list of supercomputers, we refer the read to the list of the top-500 best performing supercomputers in the world [28].

3.2.2 Classification of Parallel Architectures

Several classifications of parallel architectures exist in the literature. The most known classification is the one proposed by Flynn in 1966 [32]. This classification is based on the

organization of the flow of instructions (transmitted from a control unit) and the flow of data streams. This classification contains four classes:

Single Instruction stream Single Data stream (SISD)

The SISD class corresponds to the old serial machines where a single instruction stream is executed by one CPU using a single data stream as input. The classical machine of von Neumann is a typical example from this class of machines.

Single Instruction stream Multiple Data stream(SIMD)

SIMD machines contain several processors supervised by one control unit. For this kind of machines, all processors execute the same sequence of instructions (received from the control unit) on data that come from different data streams. The processors operate synchronously, i.e. they execute the same instruction at each unit of time.

Multiple Instruction stream Single Data stream(MISD)

MISD machines have several processors that operate on single data stream using different instruction set. Some authors consider that this class does not correspond to a realistic model of operation [17].

Multiple Instruction streams Multiple Data stream (MIMD)

In the MIMD machine mode, the processors are independent of each other and work asynchronously, and each one of them has its own control unit. like in the SIMD model, every processor in the MIMD mode uses its own data, but can additionally have its own flow of instructions.

In addition to the classification of Flynn, parallel computers can be also classified according to their memory organization either as Shared or Distributed memory machines.

Machines with *shared memory organization* contain a common memory space accessible by all processors to perform a read/write operations. this allows processors to communicate together since a data written by one processor can later be read by another one. The communication between the processors and the memory is done via an interconnection network or a bus. This organization may cause memory access conflicts which can be avoided by allowing only one processor to manipulate a given data at a given time.

For parallel machines with *distributed memory organization*, each processor has its own local private memory with an exclusive access to it. The Communication between processors can only be done by sending messages through an interconnection network that connects the processors together.

Other parallel classifications may exist in the literature. For more details the reader may refer to [29, 46].

3.2.3 Parallel Algorithms

Fast parallel algorithms are crucial to achieve a significant reduction in the running time needed to solve complex problems. The key idea to achieve this goal is the design of parallel algorithms. There are several criteria for evaluating the performance of parallel algorithms such as: acceleration, efficiency, and iso-efficiency.

The efficiency of sequential algorithms is usually measured by their running time which depends essentially on the size of their input data [20]. Let's consider X as a computational problem and n the size of its input data. We define the *speedup* (Sp) of a parallel algorithm, that solves the problem X using p parallel processors as the ratio between the time of the best sequential algorithm that solves this problem ($T_s(n)$) and the time spent by the parallel version ($T_p(n)$) to solve it. In other words, the speedup of the parallel version that solves the problem X is equal to:

$$Sp_x(n) = T_s(n)/T_p(n).$$

The *Efficiency* represents another performance measure of a parallel algorithm. It is defined as follows:

$$E_x(n) = T_s(n)/(p * T_p(n)),$$

which is equal to:

$$E_x(n) = Sp_x(n)/p.$$

This measurement gives an indication about the efficiency of using p processors in the parallel algorithm. An efficiency value equal to one indicates that the parallel algorithm runs p times faster using p processors as compared with the best serial algorithm that uses only one processor. Thus, each parallel processor accomplishes its tasks efficiently during each step of the parallel algorithm [48]. We can notice that efficiency is closely related to the number of processors. In addition to its use in comparing the sequential and parallel algorithms, the

efficiency can also be used to compare the performance of two parallel algorithms, i.e., the greater the efficiency, the better the parallel solution. As shown in figure 3.2.1, an ideal parallel speedup implies an efficiency equals to one. For some kind of algorithms and optimization problems, we can get a super-linear speedup characterized by an efficiency greater than one. This situation can appear when the parallel algorithm avoid some computations performed in the sequential algorithm. We say that a parallel algorithm is *scalable* if it remains efficient for a large number of processors.

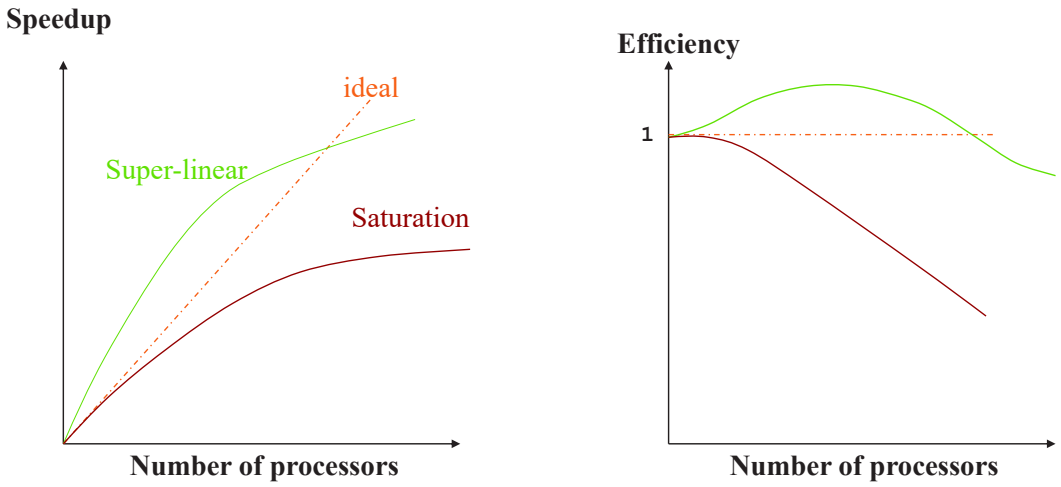


Figure 3.2.1: Different cases of speedup and efficiency.

The speedup and efficiency depend on the size of the problem and on the number of processors. By fixing the size of the problem, it is possible to study the evolution of these indicators (speedup and efficiency) according to the number of processors. The converse is also possible: by fixing the number of processors, we can study the evolution of the acceleration and the efficiency according to the size of the problem that needs to be solved. To this aim, the *Iso-efficiency* is used. This relationship can be useful for determining the ideal number of processors for a given instance of the problem or to estimate the minimum

size that a problem instance should have, in order to get some given speedup.

The performance of parallel algorithms is influenced by a set of important factors. In the following, we will present some of these factors.

3.2.4 Performance Factors for Parallel Algorithms.

The classical approach to design a parallel algorithm is to identify the independent steps in the sequential algorithm and assign them to different processors. However, this classical approach often gives inefficient parallel algorithms with a low degree of parallelism [17]. It should also be noted that some algorithms are more suitable for parallelization than others. In conclusion, we can say that extracting the parallelism from the best sequential algorithm does not necessarily lead to a good parallel algorithm.

In order to design efficient parallel algorithms, we must take into account some important parameters that will be detailed in the following.

Granularity of Parallelism.

The parallelism grain represents the average size of tasks assigned to the processors. The size of tasks is usually measured by either the execution time, the used memory space, or the number of executed instructions. Two kinds of parallelization can be identified: *coarse grain parallelism* and *fine grain parallelism*. The choice of this granularity is strongly related to the characteristics of the used parallel machine. However, in general, we consider a hierarchy from coarse grain to fine grain. *i.e.* We begin first by considering coarse grained parallelization before going to fine grained parallelization. Contrary to coarse grained parallelism, fine grained parallelism does not require a deep knowledge of the code that needs to be parallelized. However, several parts the code need to be run in parallel in order to achieve a decent speedup. In addition, this kind of parallelism (fine grained parallelism) often needs many synchronization points which may potentially affect the parallelization gain. The coarse grained parallelism requires an adequate knowledge of the code, but, a higher parallelization gain can usually be achieved since this model induces a limited number of synchronization points.

Communications

A significant part of the execution time of a parallel algorithm is dedicated to information exchange between the processors. All these exchanges represent the communication load of an algorithm. This load might influence the achievable performance according to the used communication model.

In a shared memory model, the information exchange between the processors is done through a common memory space accessible to read and write operations by all processors. In this model, the communication load of an algorithm depends on the size of the data transferred between all the processors, the memory access time, and the management of memory access conflicts induced by simultaneous read/write operations on the same data.

In a distributed memory model, communications are performed by exchanging messages, *i.e.* the processors explicitly send (resp. receive) data and instructions to (resp. from) other processors. The communication load in this model depends on the topology of the network, the available network bandwidth (the transfer capacity of the network), the size of the messages, and the number of messages sent during the execution of the algorithm. The last two elements are directly related to the structure of the algorithm and the number of used processors.

Synchronization

To ensure a valid execution of the algorithm in a parallel environment where tasks are executed simultaneously, we often need *synchronization* points to establish some order of events that are dependent on each other, so as to ensure that all the processors have completed their part of the computation at some predefined given times. In a shared memory model, synchronizations can be performed using either barriers or semaphores. If a barrier is placed in a program, no processor can continue running after that point until all processors have reached it. Semaphores are used to manage a critical area where only one processor can execute certain instructions in the algorithm. In a distributed memory model where communications are performed by sending and receiving messages, synchronization is achieved by using blocking send and receive operations. During the synchronization, processors remain inactive for a period of time that varies according to the algorithm and to the used parallelization model. Synchronization may adversely affect the parallel efficiency, one should try to minimize the synchronization points as much as possible.

Tasks Decomposition

The design of a parallel algorithm involves the distribution of computation and the data across all processors. This distribution should be as fair as possible in order to minimize the inactive time of processors during synchronizations and hence maximize the efficiency of the algorithm. Nevertheless, it could happen that the computational load of a task is not known in advance due to the variation of the number of tasks during the execution, or to the technology constraints that slow-down the execution time of certain tasks. According to Calégari [11], three task allocation strategies are generally possible. These strategies depend on the time of the allocation and on the number of tasks. If they are both determined at the compilation time of the program, then the static allocation strategy is used. Otherwise, if they both vary during the execution, then the adaptive allocation strategy is used. If the number of tasks is determined at compile time and the allocation is performed during execution, then, a dynamic allocation strategy is used. In the last two strategies, a load balancing algorithm is needed to balance the load of tasks between processors. If the computational load of the different tasks is more or less the same and the number of tasks is known in advance, then the static allocation strategy may be sufficient. If the tasks are heterogeneous, then the dynamic strategy will probably be necessary to achieve a good efficiency.

3.3 Taxonomies of Parallel Resolution Methods

In this section, we present the classification of parallel resolution methods that aim to solve optimization problems in exact or approximate way.

3.3.1 Classification of Parallel B&B Algorithm

The parallelization of B&B algorithm is well study in the literature and several classifications of this method have been proposed [24, 35, 71].

Classification of Granic et al.

Crainic *et al.* [24] classified the parallel B&B algorithm into two categories: *node-based parallelization* and *tree-based parallelization*. The *Node-based parallelization* exploits parallelism inside the B&B operations. This kind of parallelization aims to accelerate the search, without changing the general structure of the B&B algorithm, by parallelizing its operations that

are applied to sub-problems (nodes). For example, computing the lower bound in parallel for each node. The *Tree-based parallelization* consists in exploring the B&B search tree in parallel by performing the B&B operations on several sub-problems simultaneously. This model of parallelization changes the general structure of the B&B algorithm which may lead to some irregularity inducing a load balancing problem.

Classification of Trienekens et al.

Trienekens *et al.* [71] proposed two levels of parallelization namely, *low-level parallelization* and *high-level parallelization* according to their degree of parallelization. The *Low level parallelization* is similar to node-based parallelization. The general structure of the B&B algorithm doesn't change. i.e., Only some parts of it are performed in a parallel way. The exploration of the search tree doesn't change at all, it is only faster. For the*High-Level parallelization*, the B&B search tree is explored in parallel, which means that the parallelism is not restricted to a particular part of the B&B algorithm. This kind of parallelism influences the exploration process of the algorithm leading to explore some parts of the space which are not explored by the serial B&B version.

Classifications of Gendron et al.

Gendron*et al.* [35] identified three types of parallel B&B algorithms. *Parallelism type1* (*node based*): introduces the parallelism when performing the B&B operations on the generated sub-problems. For example, executing the bounding operation in parallel for each sub-problem.*Parallelism type 2* (*tree based*): consists to build the search tree in parallel by performing the B&B operations on several sub-problems simultaneously. For instance, executing the bounding operation in parallel for several sub-problems simultaneously.*Parallelism type 3* (*multi-search*): implies that several search trees are generated in parallel and each tree is characterized by the different B&B operations. In addition, the information generated when building one tree can be used for the construction of another one.

Another classification of the authors exist. This latter is based on the work pool concept which is a set of active sub-problems generated by the B&B algorithm. Two categories are identified: *Single work pool* and *Multiple work pools*. In a*Single work pool*, only one memory location is used to store the work units (sub-problems) which are manipulated by several processes concurrently to pick and insert sub-problems. In *Multiple work pools*, several

memory locations are allocated to several work pools where each work pool can be associated with one or several processes.

classification of Melab and Mezmaz

The most recent classification is the one proposed by melab and Mezmaz [57]. It represents a generalization of the classification proposed by Gendron*et al.* [35]. Four models of the parallel B&B algorithm are identified. The *parallel multi-parametric model* is a coarse-grained model based on the use of several B&B algorithms that run in parallel using different parameters. *The parallel tree exploration model* consists to simultaneously explore several sub-spaces using different B&B algorithms. *The parallel evaluation of bounds model* allows to compute the bounding of several sub-problems simultaneously inducing a different behaviour compared to the serial B&B algorithm. *The parallel evaluation of a single bound* model does not change the design of the algorithm because it is similar to the sequential version except that the bounding operation is faster.

3.3.2 Parallel Metaheuristics

The metaheuristics are efficient tools to deal with complex optimization problems by finding a near optimal solution in reasonable time. However, they need huge running time due to the importance of the search space [26]. To overcome this drawback, parallelism is used. Indeed, by using the right parameters, parallel metaheuristics can be much more robust than their sequential versions in terms of solutions quality [22]. According to Roch *et al.* [6], Parallel metaheuristics can be seen as irregular applications whose efficiency strongly depends on the choice of the granularity of the parallel algorithm and the use of load balancing techniques.

Several authors studied the classification of parallel metaheuristics, but most of the existing works in the literature focus on the parallelization of a particular metaheuristic and rare are those which bring a global and fundamental view to parallel metaheuristics [22]. On the classification of metaheuristics, we can find the work of Cung *et al.* [26] and the work of Crainic and Toulouse [22].

Classification of Crainic and Toulouse

To establish a classification, Crainic and Toulouse [22] study the common aspects of the different parallel implementations related to metaheuristics. This classification is divided

into three categories, based on the impact of the parallelization strategy on the metaheuristic structure.

The first category exploits parallelism inside an iteration of metaheuristics. This strategy is considered as a low-level parallelization in which the goal is to speed up the computations without attempting to perform a better exploration. *i.e.* The parallel metaheuristics explore the search space exactly in the same way as sequential metaheuristics. Hence, taking advantage of the computing power without fundamentally changing the algorithm. This leads to an increase in the explored search space while maintaining an overall execution time equals to or less than the sequential version.

The second category consists to divide the problem into several sub-problems and use parallelism to solve them simultaneously. The search behavior is then different from the sequential algorithm. A typical example of this category can be obtained by a master/slave model in which the master sequentially determines the initial partitions and modifies them during the search process. The slaves independently explore the search space assigned to them by the master At the end, the master assembles the partial solutions found by the slaves into a complete solution to the problem.

The third category exploits the parallelism to explore the search space simultaneously using different degrees of synchronization and cooperation. This strategy aims to perform a more complete search by involving several parallel processes that operate simultaneously using different exploration strategies. The parallel processes can communicate with each other during the search or only at the end to identify the best solution. Therefore, this category can be divided into two classes: independent approaches and cooperative approaches.

Classification of the Local Search Metaheuristics

Local search metaheuristics can be seen as a graph where the nodes correspond to solutions and the arcs connect neighboring solutions. In each iteration, local search metaheuristics move from one node (solution) to another, while avoiding as much as possible the trap of local optimum. In order to move forward, the neighboring nodes (solutions) must be evaluated. Parallel implementations of local search metaheuristics use several processors to explore this graph simultaneously. [26].

Two approaches for the parallelization of local-research metaheuristics are identified by Cung *et al.* [26]. These approaches are based on the number of steps that are performed in the neighborhood graph:*single step case* and *multiple steps case*.

In the *single-step* parallelization, a single path is created in the neighborhood graph. In each iteration of the algorithm, the search for the best neighbor is done in parallel (evaluating the neighboring solutions simultaneously using several processors). This approach aims to accelerate the passage through the neighborhood graph. It can be seen as the first category defined earlier by Crainic and Toulouse [22].

The *multiple-steps* category aims to explore multiple paths simultaneously using different processors. This strategy represents a coarse grain parallelization that aims to speedup the running time and to improve the solution quality. This category is subdivided into *independent search* and *cooperative search* depending on the information exchanged by the parallel processes. This category is very similar to the third category defined in [22].

Classification of the Parallel Tabu Search Algorithm

The tabu search algorithm is well suited for parallelization since it can be easily adapted to benefit from parallel computing architectures.

Several classifications of the parallel TS methods exist in the literature. Trienkens and Bruin [71] classified the parallelization of the TS method as *low-level* and *high-level* parallelizations. Compared to the sequential TS version, the low-level parallelization does not change the way in which the search space is explored, it is only faster. On the other hand, the high-level parallelization has a different behavior from the sequential version since the search space is explored by several parallel processes simultaneously.

Crainic *et al.* [23] introduced a taxonomy of parallel TS algorithms which remains the most widely used in the literature as of today. This taxonomy is based on three dimensions. The first one "*control cardinality*" defines the parallel TS trajectory which can be controlled by one processor (*1-control i.e.* the search space is explored in the same way as sequential TS) or distributed among several processors (*P-control*). The second dimension "*Control and communication type*" manages the communication, the organization and the synchronization between the parallel processes. It contains four degrees (*Rigid Synchronization, Knowledge Synchronization, Collegial and Knowledge collegial*) that depend on the way in which the processes handle and share the information between them. The third dimension *Search Differentiation* focuses on the starting solution and the search strategy of each parallel process. Four ways are identified: *Same initial Point, Same search Strategy (SPSS); Same initial Point, Different search Strategies (SPDS); Multiple initial Points, Same search Strategy (MPSS); Multiple initial Points, Different search Strategies (MPDS);*

For more details about the parallel TS classifications, the reader may refer to [23].

3.3.3 Performance Measures of Parallel Metaheuristics

In the following, we present how to measure the performance of parallel metaheuristics.

In the case where the parallelization belongs to the first category of the classification of Crainic and Toulouse [22] (low-level parallelization), in which the parallel algorithm follows the same exploration path as the sequential version. The benefit of parallelization in this case is the improvement of the execution time while maintaining the same obtained solution quality. Therefore, it is thus possible to measure the performance of the parallel metaheuristics by applying the standard evaluation criteria (speedup and efficiency).

For the other parallelization categories (High-level parallelization), where the search space is explored by several parallel processes leading to change the exploration process of the sequential algorithm, the benefit of parallel metaheuristics should include the quality of the obtained solutions. For that, Cung et al. [26] discussed the efficiency of parallel metaheuristics implementations, based on multiple-steps strategy using several parallel processes, by measuring the time needed by the parallel and sequential metaheuristics to find a target solution with an objective function equal the value x. The speedup (acceleration) is then given by the ratio between the time required by the sequential algorithm and the time needed by the parallel implementation that uses p processors, to find the targeted solution.

3.4 Related Works

In this section, we will focus on the existing works on solving the blocking job shop scheduling problem. In addition to the key works on the parallelization of the Branch and Bound algorithm and the Tabu search method.

Despite the large number of application areas and the economic impact of reducing the storage space in the production chain, the BJSS has been treated by relatively a few authors as compared to the classical Job shop problem.

3.4.1 Exact Resolution Methods for the BJSS Problem

Due to the complexity of the BJSS problem, only a few authors tried to solve it optimally. The most existing B&B methods for the job shop problem are based on the *one machine*

scheduling problem proposed by Carlier et *al.* [13].

In [54], Mascis and Pacciarelli studied the job shop problems with blocking and no-wait constraints. They formulated the problem by means of an alternative graph model which is a generalization of the disjunctive graph of Roy and Sussman [65]. Based on this model, they proposed an efficient B&B method in which they solved the 10×10 benchmark instances for the first time.

It is noted that our sequential version of the B&B algorithm is based on the work of [54], the only difference is in the selection process and exploration strategy. Ours is slightly better.

In [2], Ait Zai *et al.* proposed an original B&B method based on graph theory to solve the BJSS problem. The idea of its branching scheme relies on the implicit enumeration of all possible combinations on a given machine. The authors gave solutions for local instances only.

The B&B algorithms are not efficient in dealing with large instances due to their huge search space. To deal with such instances, authors exploited the computing power of high-performance computing architectures.

The B&B parallelization may be attached to the architecture of the processing machine, synchronization, granularity of generated tasks, communication between different processes and the number of computing processors [14].

Several authors have proposed to accelerate the B&B method using GPUs. Most of these works focus on solving permutation Flow Shop Problem (FSP), knapsack problem and Traveling Salesman problem.

Chakroun *et al.* [16] and [58], take the classical approach of sending nodes to be evaluated on the GPU to solve the Flow shop scheduling problem since this step takes more than 98% of the global execution time. Therefore, each GPU thread supports the evaluation of a single node of the search tree. In [8, 15] the authors extend the earlier approach to exploit the multi-core CPU processors in the generalization of sub-problems that are going to be sent to the GPU for evaluation.

In [50], Alami *et al.* proposed a CPU-GPU based B&B applied to the knapsack problem. In the proposed parallelization scheme the branching and bounding can be done either on the CPU or the GPU according to the size of the search tree. The idea behind this approach is to use a less CPU-GPU communication and better management of data-structures in the GPU memory.

In [14], Carneiro *et al.* apply the B&B to the traveling salesman problem where a pool

of nodes is sent to the GPU for evaluation. Each GPU-thread applies the branching and bounding operations to a single node which builds a local tree for each recieved node. The resulting nodes are moved back to the CPU where the promising nodes are inserted into the tree.

In [59], Melab et al. proposed multi-core and many-core parallel B&B approaches for large optimization problems. The authors propose two B&B implementations, the first one focuses on exploiting the traditional multi-core CPU processors while the seconds one is dedicated for Intel Xeon Phi coprocessors considering both native and offload modes. The reported results show that the many-core approaches (native and offload) are twice faster as compared with the multi-core CPU approach.

In [64], Riebler et al. proposed a parallel B&B algorithm exploiting the advantage of instance-specific computing on Field Programmable Gate Array (FPGA) which has proven to be highly efficient in terms of area, energy consumption, and performance. In addition, the proposed parallelization is based on work stealing strategies to ensure dynamic load-balancing between the parallel threads. The authors' approach was applied to the reconstruction of corrupted AES keys problem, the reported results show an overall speedup of 47x.

In [73], Vu and Derbel proposed parallel B&B approaches for large-scale heterogeneous distributed platforms with several distributed CPUs and GPUs. The proposed approaches address the critical issue of how to map B&B workload with the different levels of parallelism corresponding to the target computing platform. The reported results show the significant impact of the adaptive load balancing among the heterogeneous compute resources on the achieved performance.

Due to the complexity of the BJSS problem, only small instances can be solved optimally. To deal with large industrial instances approximate algorithms are unavoidable.

3.4.2 Approximate Resolution of the BJSS Problem

In [45], Hall and Sriskandarajah presented a survey on machine scheduling problems with blocking and no-wait constraints. They review the computational complexity of a wide variety of no-wait and blocking scheduling problems and describe several applications of no-wait and blocking scheduling models in manufacturing systems.

In [56], Mati *et al.* proposed a tabu search method for BJSS in which they apply the classical permutation on the critical path. The deviation (relative error) of the authors results from the optimal solutions is between 3% and 9% for the 10 jobs $\times$10 machines (10

$\times$ 10) benchmark instances.

In [54], Mascis and Pacciarelli studied the job shop problems with blocking and no-wait constraints. They formulated the problem by the means of an alternative graph model which is a generalization of the disjunctive graph of Roy and Sussman [65]. Based on this model, they developed four constructive heuristics and presented numerical results for 58 benchmark instances.

In [60], Meloni *et al.* presented a Rollout metaheuristic based on the alternative graph model [54] for the classical job shop, blocking job shop and no-wait job shop problems. This rollout metaheuristic extends iteratively a partial feasible selection of alternative arcs until a complete feasible selection (schedule) is obtained. At each iteration, all unselected arcs are scored by the heuristics presented in [54]. The authors presented a numerical result for 18 (10×10) benchmarks.

In [27], Pham and Klinkert proposed a hospital resource scheduling approach (surgical units, surgeons, nurses, patients, *etc.*). They modeled the problem as a new extension of the job shop scheduling problem called Multi-Mode Blocking Job Shop (MMBJS). They solved the MMBJS as a mixed integer linear programming problem.

In [42], Groflin and Klinkert proposed a tabu search neighborhood structure based on reversing an alternative arc in the critical path. They also extended their approach to the general job shop scheduling problem. The generalization consists to add for each operation a take-over and a hand-over times. The author's approach has allowed to improve the numerical results of [54] and [60]. In [43], the authors used the same neighborhood to take into account machine flexibility.

In [61], Oddi *et al.* proposed an Iterative Flattening Search method to solve the BJSS problem. They improved almost all the results of Groflin and Klinkert [42].

In [2], AitZai *et al.* proposed a genetic algorithm with binaries encoding to solve the BJSS. The authors reported approximate results for 40 benchmark instances.

in [55], Mati and Xie proposed a tabu search combined with a geometric approach heuristic for the blocking job shop with no swap and extend it to include resource flexibility. The authors improved all results found in [60].

In [63], Pranzo and Pacciarelli proposed an Iterated Greedy metaheuristic (IG) for BJSS based on the alternative graph model [54]. The IG algorithm applies two phases iteratively: a destruction phase in which 80% of alternative arcs in the solution are deselected and a constructive phase in which a new solution is obtained by applying AMCC heuristic in [54]

to extend a partial solution. The authors improved the most results in [42, 55, 60, 61].

The drawback of previously applied metaheuristics such as [2, 56] is the major exploration of infeasible solutions since these approaches don't take into account the specificity of the blocking constraint. The major limitation of iterative improvement meta-heuristics (IG [63], CP-Opt [61], RM [60]) lies in the random exploration of the search space which did not give good results in terms of quality for large instances, in addition to its lack of stability and performance.

3.5 Conclusions

In this chapter, we introduced the basic concepts relative to parallelism in a general way. We presented first the different classifications of parallel architectures that exist in the literature. After that, we introduced the standard evaluation metrics that are used to evaluate parallel algorithms which are *speedup* and *efficiency*. In order to design efficient parallel algorithms, we presented the most important factors that influence the efficiency of parallel algorithms which are granularity of parallelism, communications, synchronization, and tasks decomposition. In the literature, several authors have made a detailed study on how to benefit from parallel architecture to improve the performance of exact and approximate resolution methods. These study in addition to a detailed literature reviews on solving the BJSS problem were presented in the second part of this chapter.

In the next chapter, we will present the adaptation and the parallelization of the Branch and Bound algorithm applied on the BJSS problem.

4

Parallel Branch and Bound Algorithm for the BJSS Problem.

4.1 Introduction

The Branch and Bound algorithms (B&B) are well-known techniques for solving optimally optimization problems. The algorithm consists to explore all feasible solutions in a smart way by avoiding the exploration of non promising branches. The classical JSSP is known to be NP-hard in the strong sense and its search space is equal to $(n!)^m$ [45]. The BJSS problem represents an extension of the classical JSSP. Consequently, they have the same search space however, the BJSS problem appears to be even more difficult to solve [10]. Therefore, only small instances of it can be solved optimally. To deal with large benchmark instances, we propose in this chapter a new parallel B&B schemes that exploit high performance computing architectures.

The second section of this chapter presents our proposed serial B&B algorithm and its components, in addition to the impact of the exploration strategies and the selection process

on the efficiency of the B&B algorithm.

The third section describes our proposed B&B GPU-based schemes. The first scheme, named *Parallel Evaluation of the Bound (PEB)*, is a node-based parallelization exploiting the idea that the evaluation (bounding) of each node of the search tree can be calculated in parallel on the GPU. This scheme is very important in reducing the B&B execution time since the bounding phase consumes more than 90% of the whole execution time. Therefore, at each iteration of this scheme, one node is evaluated on the GPU by using several threads organized in one GPU block. The second scheme, named *Parallel Evaluation of Several Bounds (PESB)*), represents a generalization of the PEB scheme obtained by sending at each iteration a pool of nodes to the GPU for evaluation instead of one node at a time. The idea here is to achieve a good relative speedup by increasing the ratio of computation to communication on the GPU.

The forth section describes our Multi-core B&B scheme dedicated to personnel computers as well as cluster-based supercomputers. The proposed scheme in this section is based on the parallel search tree exploration (Tree-based parallelization) using Master/Worker paradigm where the master performs a breadth-first exploration, to ensure wide availability of sub-problems, and the workers perform a depth-first exploration in order to bro wse a large number of feasible solutions. Therefore, obtaining a faster improvement of the upper bound. Our proposed scheme here acts both as exact and approximate approaches by reporting respectively the optimal solution or the best explored solution during 7200 seconds. This allowed us to report approximate results for large BJSS instances in addition to the optimal solution for ten benchmark instances for the first time in the literature.

The most of the top 500 supercomputers in the world [28] are based on heterogeneous computing nodes. In order to exploit all computing resources of these nodes, the need for new parallel schemes is crucial. The drawback of the previous approaches is the fact that they exploit only one architecture at a time which represents a wast of significant computing power. To overcome this drawback, we propose in the fifth section of this chapter new parallel schemes that combine both high level (tree based) and low level (node-based) parallelization in order to benefit from both the multi-core CPU and the GPU at the same time. The proposed schemes represent a hybridization between the Multi-core CPU approach and the two GPU based approaches. This Hybridization is based on the concurrent kernels execution provided by Nvidia Multi Processes Service (MPS) which allows multiple host processes (master and workers) to execute simultaneously their kernels on the GPU. The first hybrid

scheme, named H-PEB, represents a hybridization between the Multi-core approach and the PEB GPU based approach. In this scheme, several host processes (Master and workers) can use the GPU simultaneously to accelerate the bounding of one node at a time on the GPU according to the PEB scheme. In the same way, the second hybrid scheme named H-PESB represents a hybridization between the Multi-core approach and the PESB GPU based approach. Therefore, all host processes (Master and workers) can use simultaneously the GPU to evaluate several nodes instead of one node in the H-PEB scheme.

Finally, a detailed experimentation section is given to report the results of our proposed parallel approaches.

4.2 Sequential Branch and Bound Algorithm For the BJSS Problem.

In this section, we present the adaptation of the B&B algorithm for BJSS problem.

The B&B algorithm makes an intelligent enumeration of all feasible solutions. It was introduced by Land and Doig in 1960 [51]. The branching and bounding operators represent the main two components of the B&B algorithms. The branching consists to replace the search space of a given problem by a set of smaller sub-problems. This is a recursive process, which means that each sub-problem is handled in the same way until reaching leaf nodes. The Bounding operator evaluates the ability of a given sub-problem to contain the optimal solution. There are two kinds of bounding namely the Lower Bounding (LB) and the Upper Bounding (UB). The LB is used to compute the lower bound for the evaluation of all feasible solutions in the considered sub-problem. The UB represents an upper limit of the evaluation of any sub-problem. Each solution of the problem can be considered as initial value for the UB and it is updated as soon as a new better solution is found by the B&B algorithm. To accelerate the search process, we use the elimination operator. This latter uses the bounds to eliminate the sub-problems which cannot lead to improve the current best solution found by the algorithm.

Algorithm 1 and Table 4.2.1 describe the general structure and symbols used by the proposed Branch-and-Bound algorithm.

The most effective B&B algorithms for the JSSP problem are based on the disjunctive graph model [10]. Our B&B is based on the adaptation of this approach to the blocking case (alternative graph model) [54]. Our method consists to fix an order (precedence) between

Table 4.2.1: The description of the symbols used in our B&B algorithm.

Symbol	Description
UB	Upper Bound.
$LIST$	a set of nodes (sub-problems).
$s*$	The optimal solution.
R_i	The ieme successor of node R.
LB	Lower Bound.
$LB(R_i)$	lower bound of node R_i.

every two concurrent operations. This leads to fix the corresponding alternative pair (from A), and a set of fixed alternative arcs represents a *selection*.

4.2.1 Branching

The branching process can be represented by a search tree where each node R of this tree is characterized by the couple (S_R, A_R) and represented by the graph $G(S_R)=(N, F \cup S_R)$. The set S_R denotes the set of selected (fixed) alternative arcs and A_R represents a set of unselected alternative pairs in the node R. The tree is rooted by the original problem; therefore, no alternative pairs are fixed ($|S_o|=0$). Our branching process creates two immediate successors $R1$ and $R2$ of node R by fixing the alternative pair $((i,j),(h,k)) \in A_R$. The node $R1$ (resp. $R2$) is characterized by $S_{R1} = S_R \cup (i,j)$ (resp. $S_{R2} = S_R \cup (h,k)$) and $A_{Ri} = A_R - \{((i,j),(h,k))\}$. Each successor represents the sub-search space related to its fixed alternative arc. After this, each successor is handled recursively in the same way until complete *selection* is found (leaf node), or eliminating the sub-problem and prune the tree if the lower bound value of this node is bigger than the upper bound i.e. the node can't lead to improve the current best solution.

To explore the search tree efficiently, several exploration strategies are used.

4.2.2 Search Exploration Strategies

The search strategies define the way the search tree is explored. Our goal here is to investigate the impact of the different exploration strategies on the B&B resolution time. For this purpose, four exploration strategies are tested.

Algorithm 1 Pseudo-code of the proposed Branch-and-Bound algorithm ($B\&B_{NS}$)

INPUT: UB.
OUTPUT: s^*.

```
BEGIN
1. LIST = {original problem};
REPEAT
2.  Choose a Node R and remove it from LIST;
3. IF LB(R) < UB THEN Generate successors R_i from R | (i = 1, ..., n);
4.    FOUR EACH successors R_i;
      BEGIN
5.       IF LB(R_i) < UB THEN
         BEGIN
6.            IF R_i represent one solution THEN
                 BEGIN
7.                    UB = LB(R_i);
8.                    s* = solution in R_i;
                 END
9.            ELSE LIST = LIST ∪ R_i;
         END
      END
UNTIL LIST = ∅;
RETURN s*;
END.
```

Breadth First Strategy (B_rFS)

As shown in Figure 4.2.1, the B_rFS exploration strategy consists to explore the search tree level by level. Which means that the B_rFS explores all nodes of a given level before starting the exploration of the lower levels. However with this strategy, the upper bound stays the same(without update) since we reach leaf nodes only in the last level. Therefore a poor elimination (pruning) process which induces a huge execution time.

Best First Strategy (BFS)

Unlike the B_rFS strategy, the goal of the BFS strategy is to reach leaf nodes as quickly as possible. This can be done by exploring the best node after the branching process (in term of evaluation). This allows to have more chance to reduce the upper bound. Therefore, reducing the execution time. However, in the practice and due to the existing of the blocking

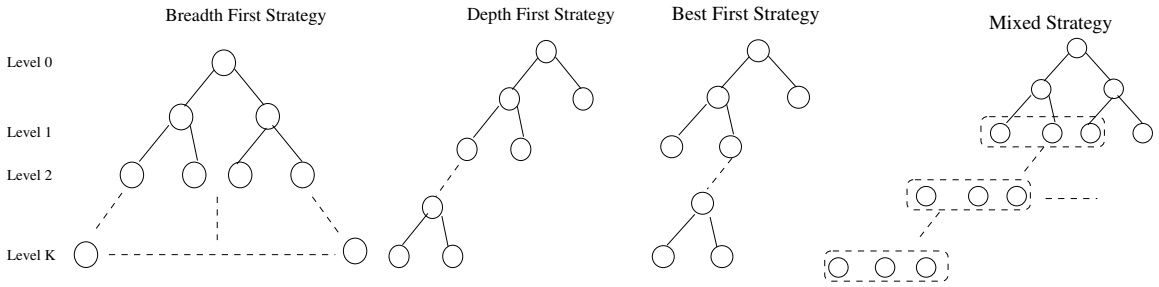


Figure 4.2.1: Exploration strategies.

constraint, most explored branches by this strategy are infeasible.

worst First Strategy (WFS)

I order to explore leaf nodes, the WFS exploration is used. It is similar to the BFS, however, instead of choosing the best after the branching process, we choose the worst first. The advantage of this strategy, as compared with BFS, is the huge number of feasible solutions explored in short time. This leads to improve the UB and eliminates a large number of branches. Thus, improving the running time.

Mixed Exploration Strategy (MES)

This exploration strategy represents a combination of *WFS* and B_rFS strategies. It is similar to the WFS strategy, but instead of exploring only one node at a time we explore several nodes at a time. This strategy is well adapted for GPU architecture. The goal here is, on one hand, to have an efficient pruning process by exploring leaf nodes, and on the other hand to increase the GPU occupation.

Figure 4.2.2 shows the graph representation of a BJSS instance with two jobs and two machines. The figure shows the existence of two alternative pairs. The first one is ((2,4), (4,2)) between operation 2 and operation 4 and the second one ((2,3), (4,1)) is between operation 1 and operation 3.

Figure 4.2.3 shows the search tree related to the BJSS instance in Figure 4.2.2. As we can see from the figure, the depth of the search tree is equal to the number of unselected pairs in the root node. At each level, one unselected pair is fixed. For instance, in the first level, the pair (2,4), (4,2) is fixed. Hence, creating two successors. After that, each successor is

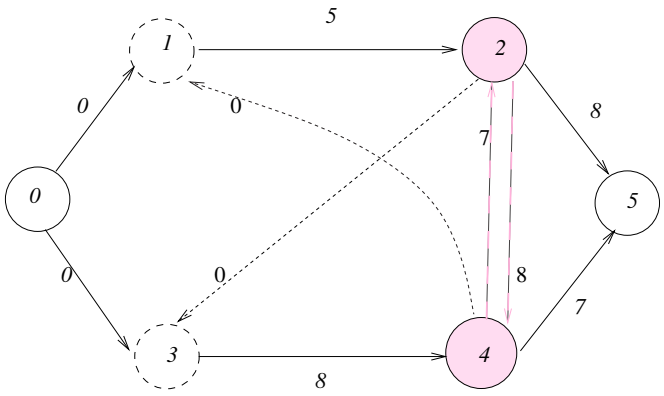


Figure 4.2.2: Alternative graph for BJSS instance with two jobs and two machines.

handled in the same way until reaching leaf nodes or prune the tree if the node contains an infeasible selection or the lower bound is bigger than the upper bound.

Moreover, Figure 4.2.4 shows the graph representation of the optimal solution obtained from the search tree in the Figure 4.2.3. This solution corresponds to the selected alternative arcs {(2,4),(2,3)} and its makespan (evaluation) is equal to 20 representing the longest path in the graph representation of the solution.

4.2.3 Proposed Selection Process

We said earlier that the branching process creates two immediate successors $(R1, R2)$ of node $R\,(S_R, A_R)$ by fixing an alternative pair $((i,j),(h,k)) \in A_R$: $R1 = (S_R \cup (i,j), A_R - ((i,j),(h,k)))$ and $R2 = (S_R \cup (h,k), A_R - ((i,j),(h,k)))$. The objective here is to find the appropriate unselected alternative pair to fix. The choice of the unselected alternative pair is very critical for the B&B algorithm since its efficiency depends on it. Indeed, a good choice of the unselected alternative pair allows quickly to see whether a branch can improve the current best solution or not; therefore, prune it. Hence, reducing the size of the search tree and avoid unnecessary computations.

In the following, we describe our proposed selection process. Let us define P_R as a set of critical operations representing the longest (critical) path in $G(N, F \cup S_R)$ and A_{P_R} the set of unselected pairs, chosen from A_R, having direct impact on operations in P_R. More precisely for each pair $((i,j),(h,k)) \in A_R$, if $(j \in P_R$ or $k \in P_R)$ then $A_{P_R} = A_{P_R} \cup ((i,j),(h,k))$. Our selection process consists to choose from A_{P_R} the alternative pair $((i,j),(h,k)) \in A_{P_R}$ that increases the most the value $Vpair = max\{(l(0,i) + a_{ij} + l(j,n)), (l(0,h) + a_{hk} + l(k,n))\}$. The idea here is

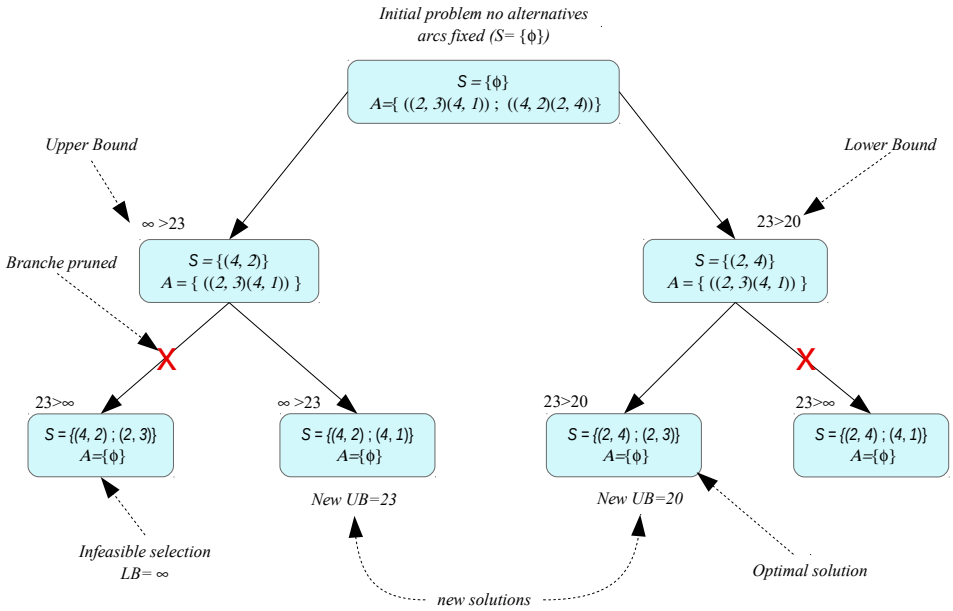


Figure 4.2.3: Search tree for the BJSS instance in Figure 4.2.2.

to rapidly increase the evaluation of sub-problems which allows us to reach leaf nodes or rapidly eliminate the non promising branches. Hence, reducing the size of the search tree and improving the running time.

Immediate Selection

The immediate selection represents several techniques leading to accelerate the B&B algorithm by reducing the number of branching necessarily to obtain the optimal solution. Therefore, reducing the size of the search tree. It represents about 15% of the global execution time of the B&B method.

Given a sub-problem R with feasible selection S_R and a set of unselected pairs A_R, we check for each unselected alternative pair $((i,j),(h,k)) \in A_R$ the following two rules:

Rule 1: If $l(o,h) + a_{hk} + l(k,n) \geq UB$ then S_R=$S_R \cup (i,j)$. This rule expresses the fact that adding the arc (k,h) (resp. (i,j)) to S_R will produce a sub-problem with a lower bound greater than the upper bound, consequently the arc (i,j) (resp. (h,k))is added to S_R.

Rule 2: If $l(k,h) + a_{hk} > o$ then S_R=$S_R \cup (i,j)$. This rule expresses the fact that adding the arc (k,h) (resp. (i,j)) to S_R will produce a sub-problem with infeasible selection, consequently

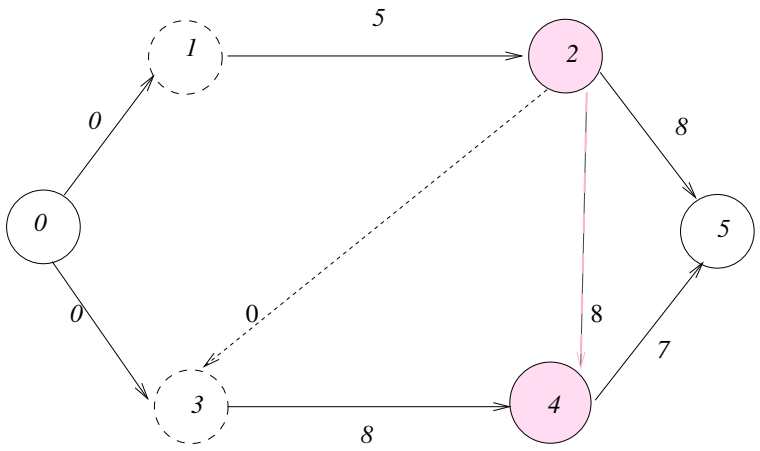


Figure 4.2.4: Alternative graph of the optimal solution (*Cmax=20*).

the arc (i, j) (resp. (h, k)) is added to S_R.

If both alternative arcs $(i, j), (h, k)$ produce infeasible selection or non promising sub-problems, we eliminate the sub-problem R. Moreover, other rules exist in the literature. For more information, the reader may refer to Mascis and Pacciarelli [54].

4.2.4 Upper Bound

A large number of sub-problems can be eliminated if a good upper bound is considered from the beginning. For that, we set initially the upper bound to the results found by [42, 43]. If a node contains an infeasible selection, which means that the corresponding graph contains a positive length cycle, we set its lower bound to ∞ i.e. we prune the branch.

4.2.5 Lower Bound (Evaluation)

The lower bound represents the most important part of the B&B algorithm since it represents 85% of the global execution time. Therefore, it is crucial to be efficient and optimized. The LB used in our case is based on the *one machine scheduling problem* used by *Carlieret al.* [13] to solve optimally the JSSP. This lower bound gives good bounds for our problem without any adaptation to the blocking case. Given a sub-problem with feasible selection S. For each operation o_i we say that $r_i = l(0, i)$ is the head of operation o_i, and $q_i = l(i, n)$ is the tail of operation o_i. Let us define a bloc V_l as a set of concurrent operations on the same machine M_l $\{l = 1...m\}$. For each block V_l, a lower bound for the *one machine scheduling problem* is calculated on this machine with the following formula: $h(V_l)$ =Min $r_i + \Sigma p_i$+Min q_i and the

lower bound for the problem is equal to the maximum of $h(V_l)$. This lower bound is similar to the Makespan of this sub-problem obtained by adjusting head and tail structures [13].

The sequential B&B takes huge time to solve small instances. To accelerate this method in order to deal with large problem instances, we propose in the following several parallel schemes that exploit HPC architectures.

4.3 GPU-based Parallelization Approaches

In this section, we describe our proposed GPU-based parallelization schemes for the B&B algorithm.

The GPU architectures are based on *SIMT* (Single Instruction, Multiple Threads) paradigm. According to this paradigm, the same program called *kernel* is executed simultaneously by a set of parallel threads with different data. The threads are organized according to a grid of thread-blocks hierarchy specified in the kernel call. The grid represents a set of thread-blocks. Threads of the same block can cooperate by using a private shared memory and barrier of synchronization. Threads can access multiple memory spaces: ***constant memory*** and ***texture memory*** are read-only cached memory accessible by all threads. The ***global memory*** is a read-write memory, also accessible by all threads. Unlike the global memory the ***shared memory*** is a cached memory accessible only by the threads of the same block [25].

Parallel Evaluation of the Bound (PEB)

We have seen previously that the evaluation process and the immediate selection consume together more than 90% of the global execution time. Therefore, it is crucial to accelerate this phase in order to reduce the B&B execution time.

In the following, we present our proposed node-based parallelization scheme for the B&B algorithm exploiting GPU-based architectures. The proposed scheme, referred to as *Parallel Evaluation of the Bound* (PEB), exploits the idea that the evaluation and immediate-selection for each node can be done in parallel using several threads.

As shown in Figures 4.3.2 and 4.3.1, this approach uses the same design as the sequential B&B algorithm with the exception that the evaluation (bounding) of each node is done in parallel on GPU. Consequently, it explores the search space in the same way as a serial B&B

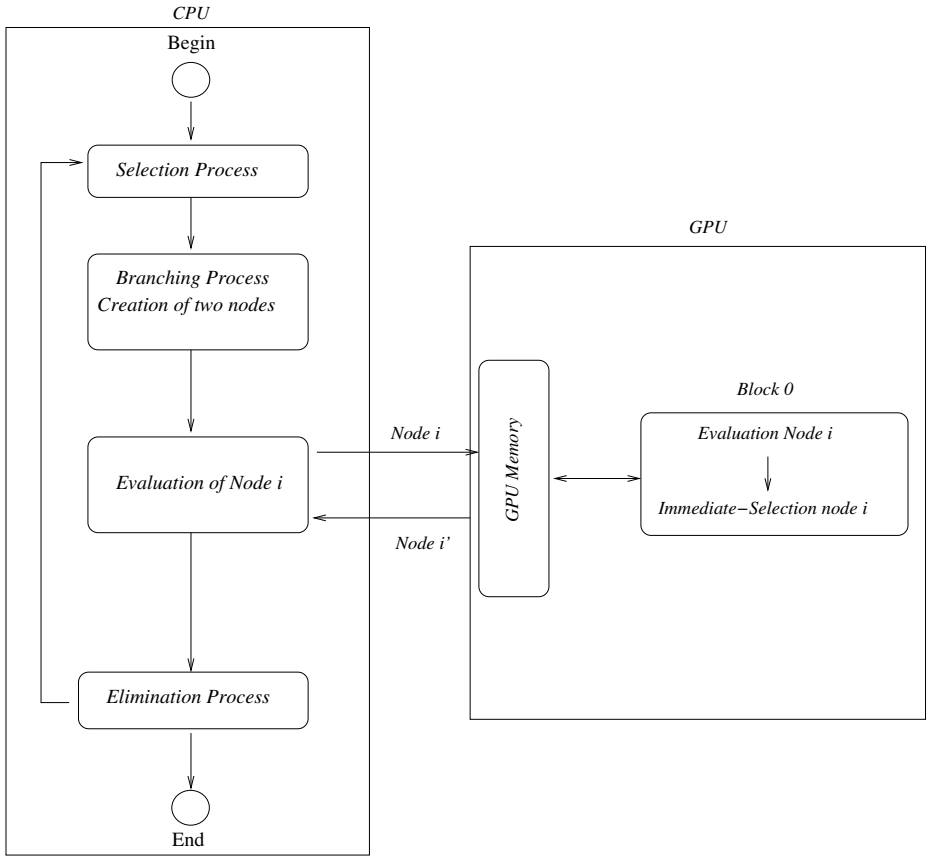


Figure 4.3.1: Parallel Evaluation of one Bound (PEB scheme).

algorithm, but faster. As already presented, each node of the search tree represents a graph of $n \times m$ operations. The bounding process consists in updating the head and tail values for each operation in the graph. The parallel *PEB* scheme is based on the idea that each GPU-thread updates the head and tail values for a single operation in the graph. This scheme exploits the fact that the updating process can be done independently for each operation. Therefore, the GPU block size is equal to the number of operations in the graph ($n \times m$). As shown in Figure 4.3.2, at each iteration, only one node is sent to the GPU for evaluation and immediate selection using one thread-block. The bounding begins by copying the head and tail vectors to the shared memory, *i.e.* each thread copies the head and tail values relative to its *id*. After that, each thread updates the head and tail values (H[i], T[i]) for the operation relative to its id (i) using respectively the head of its predecessors and the tail of its successors.

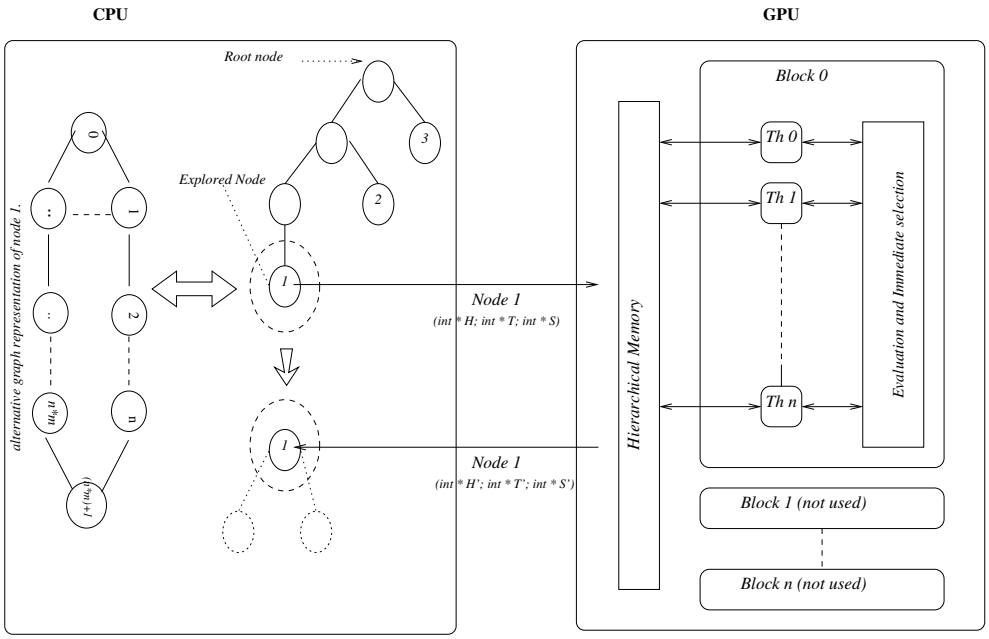


Figure 4.3.2: GPU evaluation of a single node.

$$H[i] = \text{Max } \{H[r] + p_{ri}\} \ / \ r \in Pred[i].$$

$$T[i] = \text{Max } \{T[r] + p_{ir}\} \ / \ r \in Succ[i].$$

At the end of this computation, each thread waits for the other threads of the block using a barrier of synchronization to ensure the visibility of the new head and tail values to all threads of the block, which is important to have a valid update process. The work is repeated several times until there is no update of the head or the tail values for all threads or an infeasibility is detected. After the end of the bounding process, each thread computes the immediate selection for a set of unselected alternative pairs using the new head and tail values. Finally, the new results are copied back to the global memory in order to be sent back to the CPU. These results are used by the branching and elimination operators.

As can be seen in Figure 4.3.2, a single block is used on the GPU to evaluate one node, while the other blocks remain idle. The weakness of this solution lies in the under-utilization of the GPU capacity. Thus, a waste of a significant computing power. To overcome this drawback, we propose a second level of parallelization.

Parallel Evaluation of Several Bounds (PESB)

In this section, we propose a second level of parallelization to increase the GPU occupation. This level represents a generalization of the first scheme (PEB) called, *Parallel Evaluation of Several Bounds* (PESB). The goal here is to increase the GPU occupation by generalizing the idea of the first level (Bounding is faster) to exploit more efficiently the GPU computing power. For that, a mixed exploration strategy is used, i.e. at each iteration, a pool of nodes is sent to the GPU for evaluation and immediate-selection where each GPU-block supports the evaluation of a single node. After that, the new results, for each node, are sent back to the CPU to be used by the selection, branching, and elimination operations as shown in Figure 4.3.3.

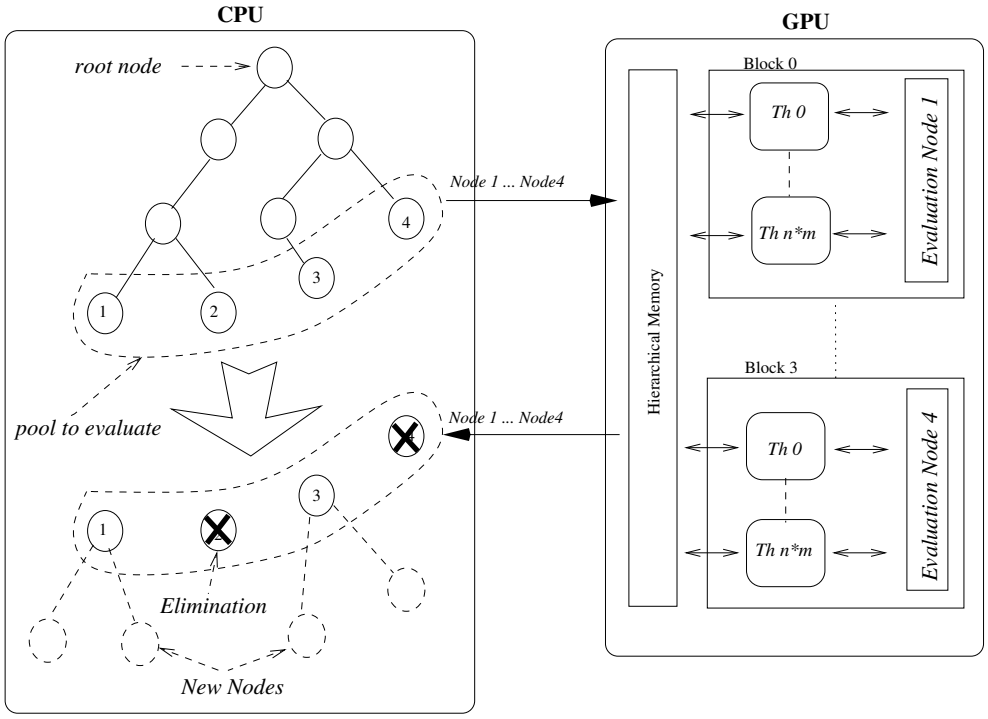


Figure 4.3.3: GPU evaluation of several nodes.

4.3.1 Work-pool Management

In both PEB and PESB schemes, there is only one work pool and one instance of the B&B algorithm that manages it and explores the search tree. In the PEB scheme, the work pool

is explored in the same way as a serial B&B approach. In the PESB scheme, the exploration strategy changes from the serial approach. Indeed, the mixed exploration strategy, explained earlier, is used in the PESB scheme. Hence, going depth-first by using k-nodes from the ones recently added to the work-pool. The number of nodes depends on the size of the instance and the used GPU resources by each node.

4.3.2 Data Organization in GPU Memory

As we have already seen, five data structures are used for the bounding of each node on the GPU. The vectors Head (H $[m * n]$), Tail (T $[n * m]$) and alternative pairs (S $[nbpair]$) are sent from the CPU to the GPU. Therefore, they are stored in the global memory of the GPU. The matrices *Succ* and *Pred* are also stored in the GPU global memory. These two matrices are calculated on the GPU using the vector S as an initialization for the bounding. To accelerate this initialization phase, the charge is divided across all the threads of the block.

The access to the global memory is much longer than the shared memory, but the latter is smaller compared to the global memory. The number of blocks that can run in parallel on each Streaming Multiprocessor depends on the amount of shared memory used by each block. Therefore, we decided to use the shared memory only for the Head and Tail vectors since the number of access to these vectors is very high.

4.4 Cluster-based Parallel Search Tree Exploration

The fact that each node of the B&B search-tree can be explored independently amplifies the parallelization of the algorithm. The only global information in the algorithm is the value of the upper bound. In this section, we present our parallel tree exploration scheme for the BJSS problem. This approach is viewed as both exact and approximate methods by reporting the optimal solutions for small instances and approximate solutions for large BJSS instances. Indeed, finding the optimal solution for such instances takes hundreds of years. For this reason, we adapted the B&B algorithm to report the approximate results. The proposed parallel B&B algorithm is dedicated for cluster-based architectures, workstations, super-calculators, etc. This approach (See Figure 4.4.1) is based on the master/worker paradigm. The master and worker processes are launched on the different cores of the cluster. The master is the root process and there is only one instance of it in the system. There are one

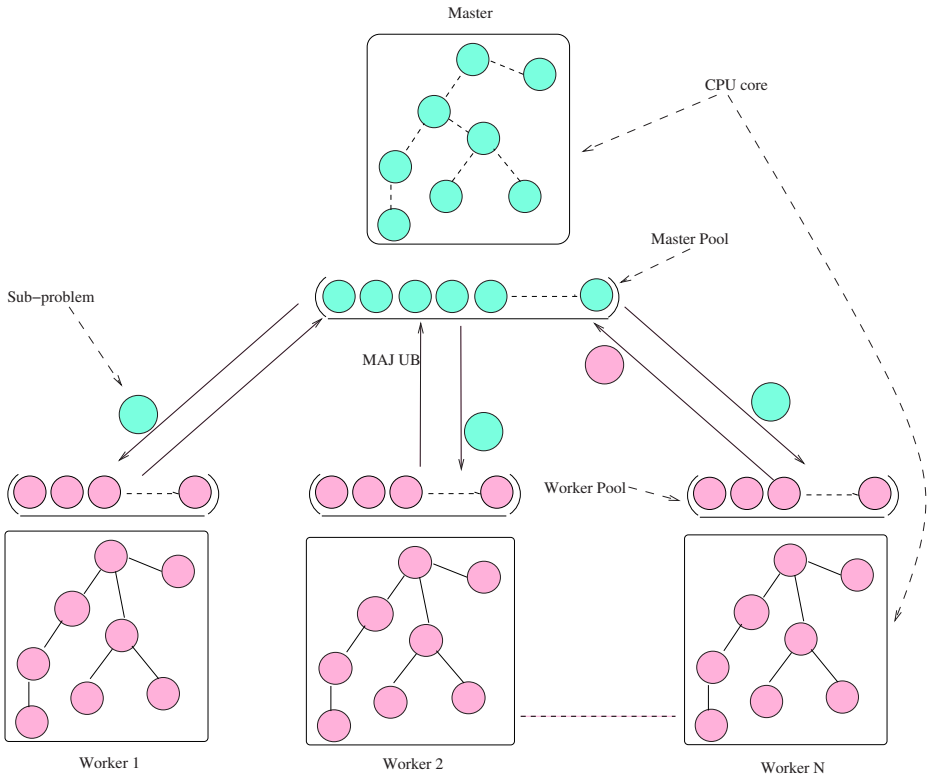


Figure 4.4.1: Global architecture of the proposed parallel B&B algorithm.

or more instances of the worker processes launched on one or multiple cores of the cluster.

4.4.1 Parallel B&B Functioning

In the following, we describe the way in which the parallel B&B approach explores the search space. A work pool represents a set of active sub-problems. In this proposed approach, there are two types of work pools: a unique global work pool managed by the master and several local work pools owned by the different workers. As shown in Figure 4.4.1, each worker manages its own work pool which means that a collegial strategy is considered. The master initializes the search tree by creating the root, launches its own B&B algorithm which generates a set of active sub-problems stored in the global (master) work pool. After that, the master wakes up the blocked workers by sending to them a sub-problem from the global work pool. After receiving nodes from the master, the workers launch their own B&B algorithm. During the search, the local pools evolve continuously and when a work pool becomes empty, the corresponding worker sends a request to the master and waits for a sub-problem. The exploration of the search-tree is done simultaneously by the different workers and the results given by a worker can influence others; therefore, our parallel B&B algorithm can be classified as a parallelization type three. The parallel exploration of the search-tree allows rapidly to achieve fine-grained tasks and improves the upper bound, which is not easy to achieve by serial version of the algorithm.

4.4.2 Work Pool Management

In this parallel scheme, several workers explore simultaneously the search tree and each one of them has its own local work-pool obtained by exploring the sub-problem (node) received from the Master. It is mentioned that the work pools are totally independent of each other. During the search, the work pool evolves continuously and when it becomes empty, the worker sends a request to the master to send it back a sub-problem. The master satisfies the request if its pool is not empty. When the global work pool is empty, the master sends a request to all workers to send it back a sub-problem. The distribution of the work via the master represents our load balancing mechanism to prevent the situation where some workers have finished while others are still running. Two states are then reserved for each process (blocked or active). Each time the global work pool is empty, the master checks the state of all workers. If all the workers are blocked, then the master ends the calculation and frees the

workers.

4.4.3 Exploration Strategies

In our proposed parallelization, two different exploration strategies are used according to the role of the process: breadth-first search for the master and worst-first search for the workers. As mentioned before, the master contains the initial problem (root node) performs the branching and bounding operators according to B_rFS exploration model. i.e., The master explores all sub-problems of a level before starting the exploration of lower levels (Figure 4.2.1). Initially and according to B_rFS model, the master explores the first k levels of the original search tree to generate at least n sub-problems assigned to the different workers to complete the exploration. The sub-problems in this strategy contain enough tasks for workers. Thus, reducing the communication load of the parallel approach. After this step, the master continues its own exploration using an adaptive exploration strategy, i.e., it uses the B_rFS exploration if its work pool size is below a given value. Otherwise, it uses the same exploration strategy as the workers. The goal of this strategy is to ensure wide availability of sub-problems for workers without reaching memory saturation. In addition, the master provides a communication link between the workers to share the new values of the upper bound. Since the role of workers is to complete the exploration, they perform the WFS exploration to reach more quickly feasible solutions. Therefore, having more chance to improve the upper bound or quickly eliminate the branches if the lower bound is greater than the upper bound. A worker which finds a better solution than the current best one broadcasts the new value to all workers to ensure an efficient branching process.

4.4.4 Communication

The communication and the synchronization between the different processes are necessary to ensure a valid execution of the B&B algorithm. Since the parallelization is done on distributed nodes interconnected by a high-speed network (InfiniBand), the communication between the different processes is done by Message Passing Interface (MPI). As we said, the only global information in the algorithm is the value of the upper bound. So when a worker improves the value of upper bound, it broadcasts the new value to all workers via the master. The advantage of this approach is the wide range of machines that can be supported by this approach since the communication load between the workers and the master is very small.

The traditional parallelization approaches do not exploit all computing resources of a machine since most of today supercomputers are heterogeneous. Therefore, the need for new parallelization schemes is essential.

4.5 Hybrid Parallel B&B Approaches

The weakness of the previous approaches lies in the under-utilization of either the GPU or CPU resources since they exploit only one architecture at a given time. Thus, a waste of significant computing power. To overcome this drawback, we propose in the following two new hybrid approaches that combine both high and low levels parallelization models.

In this section, we present two hybrid Master-worker/GPU based approaches. These approaches represent a hybridization between the high-level parallelization (the multi-core approach) and the low-level approaches that exploit the GPU architecture. This hybridization is based on the Nvidia Multi Processes Service (MPS) which is a client-server runtime implementation of the CUDA API used to increase the overall GPU utilization. Without MPS, only one host process can use the GPU at a given time; therefore, it potentially might underutilize the GPU resources. To overcome this problem, Nvidia provides the MPS to enable multiple host processes (MPI processes) to use the Hyper-Q capability on the Nvidia Kepler GPUs. Hyper-Q allows a single host process to execute multiple CUDA kernels concurrently on the same GPU. As we can see in Figure 4.5.1, the MPS consists of several components: the control daemon process is responsible for starting and stopping the MPS server, as well as coordinating the connections between clients and the server [21].

The server process provides the connection between clients and the GPU which allows concurrency. Each process (server, clients) has its own CUDA context for its GPU operations (send, receive, computation.), i.e., each process uses the GPU as it is the only one using it. When the MPS client connects to the control daemon, the latter creates the MPS server if no server is active. After that, the client proceeds to connect to the server which creates a context for the client GPU operations. Note that all communications between MPS clients/server and MPS control daemon are done using a named Pipe. Furthermore, Figure 4.5.2 shows how to use the Nvidia MPS to run MPI applications. For additional details about Nvidia MPS, the reader may refer to [21].

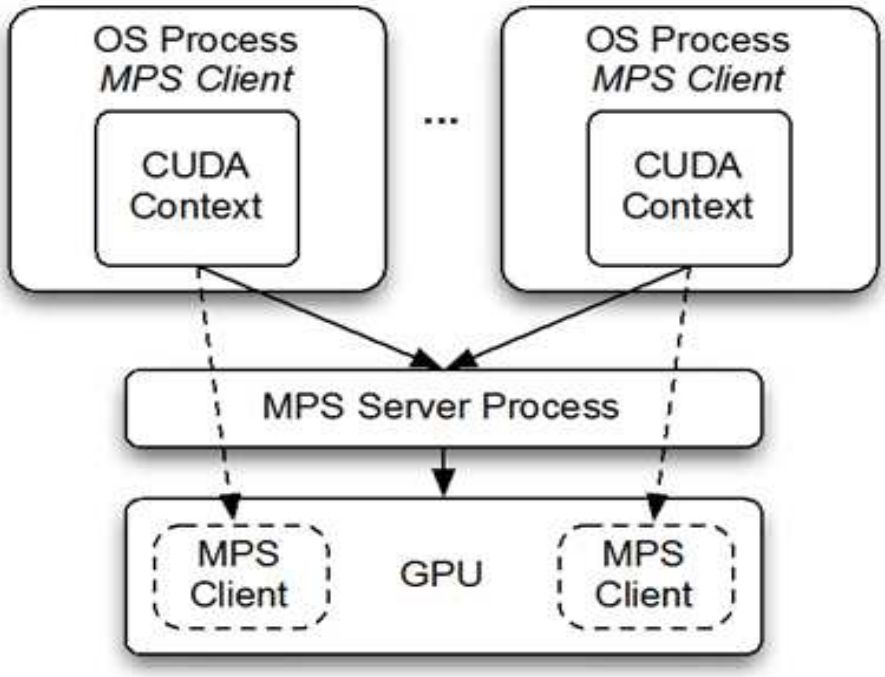


Figure 4.5.1: MPS components.

```
mkdir /tmp/mps /tmp/mps-log
export CUDA_VISIBLE_DEVICES=0                          # SELECT GPU 0.
export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps                # NAMED PIPES
export CUDA_MPS_LOG_DIRECTORY=/tmp/mps-log             # LOGFILES
nvidia-cuda-mps-control -d                             # START THE DAEMON
unset CUDA_VISIBLE_DEVICES
mpirun -x CUDA_MPS_PIPE_DIRECTORY=/tmp/mps -np 35 ./BB
export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps     # SELECT THE LOCATION OF MPS DAEMON
echo quit | nvidia-cuda-mps-control                    # STOP MPS DAEMON
rm -rf /tmp/mps /tmp/mps-log
```

Figure 4.5.2: Running MPI application using MPS.

4.5.1 Hybrid Parallel Evaluation of the Bound (H-PEB)

We propose in this section a hybridization of the tree based parallelization, that exploits the multi-core CPUs, and the GPU PEB approach. The proposed approach here, named H-PEB, aims to use a hybrid scheme that combines different parallelization levels in order to exploit both multi-core CPUs and GPU at the same time. The goal here is to increase the overall GPU occupation thus, improving the runtime. The hybrid approach is based on concurrent kernels execution provided by Nvidia MPS in devices of compute capability 2.x and higher. The maximum number of kernels that a device can execute concurrently varies between 16 and 32 according to device compute capability [21].

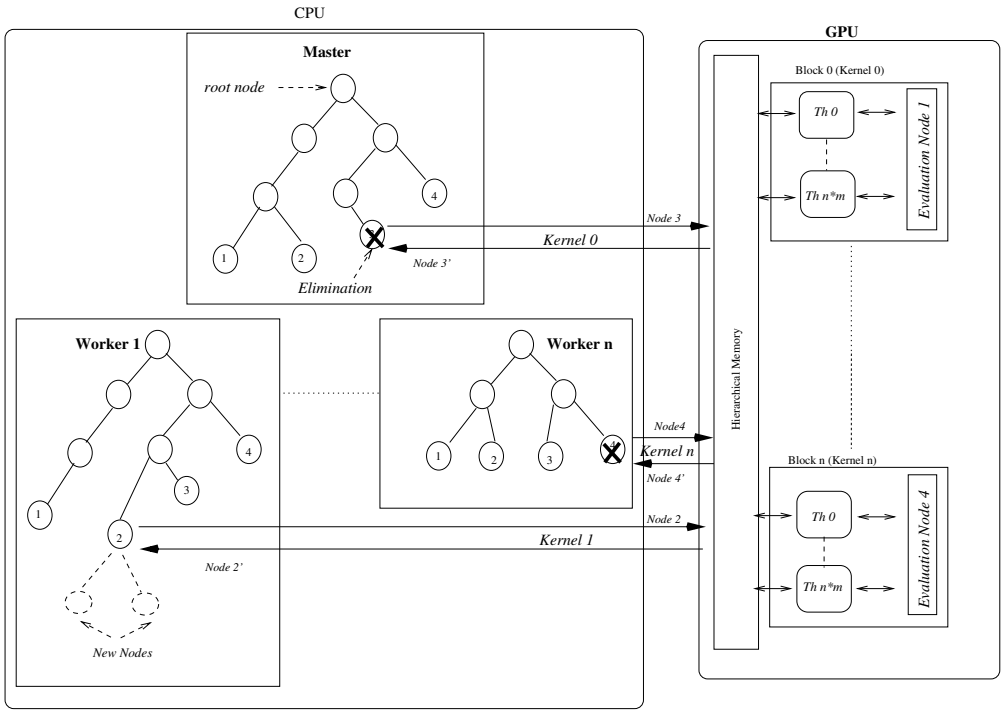


Figure 4.5.3: Hybrid Multi-core CPU/GPU approach.

In the H-PEB scheme, we have several host processes that explore the search tree (tree based parallelization) in parallel using the master/worker paradigm. As explained earlier in section 4.4, both master and workers have their own B&B instances running on different CPU-cores. The hybridization consists that each B&B instance (master or workers) uses the GPU to accelerate the bounding of one node at a time according to the PEB scheme. It is noted that each host (MPI) process launches its own kernel in the default stream and

it is up to the MPS server to manage and execute the kernels in parallel by using different CUDA-Streams. The advantage of our hybrid approach based on concurrent kernel execution is the occupation of the GPU over time. i.e., At each moment, our hybrid approach can have simultaneously several workers executing instructions on the GPU while others perform data-transfer from/to the GPU and yet others apply the selection and elimination operators on the CPU. Furthermore, Figure 4.5.3 shows the general scheme of the H-PEB approach. In this scheme, we have several B&B instances that build the search tree in parallel. therefore, each B&B instance has its own local work pool managed exclusively by the corresponding worker (resp. master) where it chooses the node sent to the GPU and performs the necessary update after each iteration. The GPU memory organization in this hybrid approach is the one used in the PEB scheme.

Using this approach, we have been able to increase the occupation of the GPU and reduce the runtime. However, The GPU is not yet fully occupied since we have a physical limit of the number of parallel processes that the MPS server can manage. For this reason, we propose in the following another hybrid parallel approach to fully occupy the GPU.

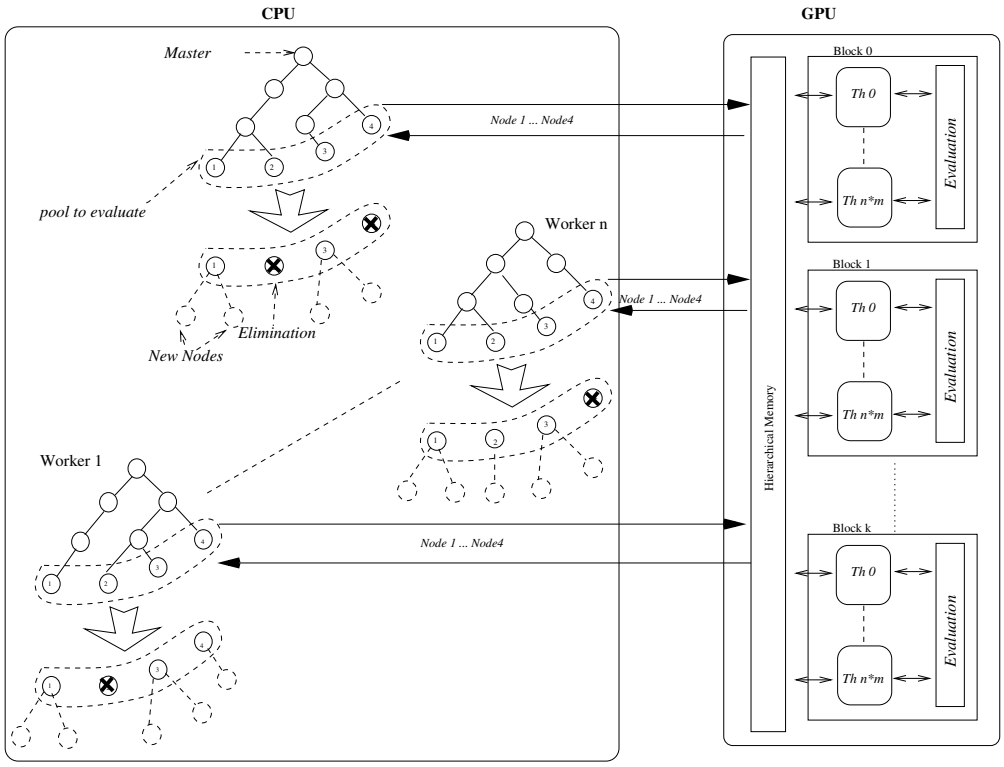


Figure 4.5.4: Hybrid Parallel Evaluation of Several Bounds (*H-PESB*).

4.5.2 Hybrid Parallel Evaluation of Several Bounds (H-PESB)

In order to fully occupy the GPU, we propose in the following a second hybrid approach called Hybrid Parallel Evaluation of Several Bounds (*H-PESB*). This approach represents a hybridization between the tree based parallelization approach (multi-core version) and the GPU-based *PESB* approach.Similairely to the H-PEB approach, the H-PESB approach is also based on the Nvidia MPS tool which makes possible that several MPI processes use the GPU at the same time. This approach is also similar to the H-PEB scheme, except that at each iteration, each host process sends a pool of nodes to the GPU for evaluation and immediate-selection instead of one node at a time. i.e., Several GPU-blocks are allocated for each process and each one of them supports the evaluation of a single node. Therefore, each thread of the block updates the head and tail values for one or several operations as we explained earlier in the *PESB* approach. At the end, the bounding results of nodes are sent back to the CPU to be used by the corresponding host process for selection, branching and elimination operations. As shown in Figure 4.5.4, The master ho contains the root node

begins by dividing the search space between the workers. After that, each worker explores its search space independently from the others, using its own B&B instance, and uses the GPU to evaluate several nodes at a time. As explained previously, The master and workers use the GPU simultaneously thanks to the Nvidia MPS that allows concurrent kernels execution while respecting the availability of GPU resources. we don't have a memory conflict problem between the host processes because each one uses its own context for its GPU operations. i.e., Each host uses the GPU as it is the only one using it.

It is noted that each B&B instance manages its work-pool according to the mixed exploration strategy explained in the PESB parallel scheme.

4.6 Experiments

In this section, computational results and discussions are given using benchmarks obtained from the classical JSSP by dropping the infinite buffer capacity constraint and replacing it with a zero buffer capacity. In order to optimize our serial B&B algorithm, we begin this section by exploring the important impact of the exploration strategy, selection process, and immediate selection on the complexity of the B&B algorithm. After that, we report the benefit of using parallel architectures to reduce the algorithm complexity via our proposed GPU-based and hybrid parallel schemes. All the experiments in this chapter that measure the execution time and the number of explored nodes where performed using the BJSS problem with Swap allowed (BWS).

At the end of this section, we report the ability of using our high-level parallel B&B algorithm (cluster-based version) as an approximate method for both BJSS cases (Blocking with swap (BWS) and Blocking no-swap (BNS)).

4.6.1 Optimizing the Serial B&B Algorithm

In the following, we report the importance of optimizing the B&B algorithm before taking advantage of parallel architectures. For that, we use small Lawrence instances [52] for which the optimal solution can be found easily.

Table 4.6.1 summarizes the results of our B&B algorithm using different exploration strategies. The first column reports the name of the instance. The second column reports the upper bound used by our B&B algorithm to compute the optimal solutions of these instances. The

Table 4.6.1: The impact of the exploration strategies on the B&B complexity (Execution time).

Instance	UB	Optimal solution	B_rFS	*BFS*	*WFS*	*DFS*
La01	1000	793	>3600	12.7	2.49	2.74
La02	1000	793	>3600	26.0	10.73	10.79
La03	1000	715	>3600	14.3	9.18	7.49
La04	1000	743	>3600	13.9	5.41	7.14
La05	1000	664	>3600	22.9	6.95	6.85

third column reports, for each instance, the optimal solution reached by most exploration strategies. Columns B_rFS, *BFS*, *WFS*, and *DFS* show the time needed by our B&B algorithm using respectively breadth-first, best-first, worst-first and depth-first exploration strategies.

The first observation from Table 4.6.1 is the importance of choosing wisely the exploration strategy since the efficiency of the B&B algorithm depends on it. Indeed, the choice of the exploration strategy varies from one problem to another because of the nature and the specificity of each problem. Even for such small instances, the B&B algorithm using the B_rFS could not reach the optimal solution. This strategy induces a poor pruning (elimination) process since it reaches leaf-nodes only in the last level, therefore, the UB stays the same which leads to spend a long time in exploring unpromising branches. The other exploration strategies reach the optimal solution with different complexity. Unlike most scheduling problems, the *WFS* and *DFS* outperform the results of the *BFS* which is the base of the most literature approaches. These two strategies are two to five times faster compared to the *BFS*. Indeed, most explored branches by this latter strategy lead to infeasible solutions. Hence, a low ratio of explored leaf-nodes to explored nodes inducing a slow decrease in UB compared to the *WFS* and *DFS*.

For this reason, we have chosen the *WFS* as the exploration strategy of our B&B algorithm.

Table 4.6.2: The impact of the selection strategy on the B&B running time.

Instance	B&B using guided selection strategy		B&B using a random selection strategy	
	Time	NB-explored nodes	Time	NB-explored nodes
La04	6.4	94508	>3600	>41652314
La05	6.6	107377	>3600	>54706752

Table 4.6.2 shows a comparison between the B&B algorithm using our proposed selection strategy and the B&B algorithm using a random selection strategy. The selection strategy consists to choose the appropriate unselected alternative pair in order to perform the branching process on it. Table 4.6.2 reports the time and the number of explored nodes by each selection strategy. The table shows the huge impact of the selection strategy on the algorithm complexity. Indeed, regardless the huge number of explored nodes, the B&B algorithm using a random selection strategy fails to reach the optimal solution even after one hour of execution time. While the B&B algorithm using our proposed selection strategy reach the optimal solution after only six seconds. Our selection strategy is based on choosing the unselected pair that increases the most the makespan which allows us to quickly know whether a branch is promising or not. Thus, avoiding a lot of unnecessary branching.

Table 4.6.3: The impact of the immediate selection on the B&B running time.

Instance	B&B with immediate selection		B&B without immediate selection	
	Time	NB-explored nodes	Time	NB-explored nodes
La04	6.4	94508	55.8	773712
La05	6.6	107377	60.8	930286

Table 4.6.3 shows the time and number of explored nodes by our B&B algorithm with (resp. without) using the immediate selection process. Table 4.6.3 shows the good impact of the immediate selection process on the B&B complexity. i.e., The B&B algorithm using the immediate selection process is up to ten times faster compared to the B&B algorithm without immediate selection. The improvement in complexity when using the immediate selection process is the result of reducing the number of branching needed to obtained the optimal solution leading to reduce the number of explored nodes and thus, the running time.

In all following experiments, we call the B&B algorithm using WFS, guided selection process, and immediate selection an optimized B&B algorithm.

4.6.2 Results of the GPU-based and Hybrid Approaches

In the following, we report the ability of our proposed low level and hybrid approaches to accelerate the B&B execution time. For that, largest Taillard instances [69] have been used. The size of these instances varies between 15×15 and 100×20. The experiments have been

carried out using a workstation that contains Intel Xeon E5640 CPU with four CPU-cores and 2.67 GHz clock speed and Nvidia Tesla K40 with 2280 cuda cores and 12 GB GDDR5 of global memory. The approaches have been implemented using C-CUDA 7.0 and C++ languages, in addition to Message Passing Interface (MPI) [44] as a communication tool between the parallel processes.

All reported times here represent the average execution time needed to explore an equal number of nodes for each benchmark size. In our case, this number is fixed to 700,000 nodes, which is an acceptable number since an optimized serial B&B implementation takes 19 hours to complete this number of nodes. For the 100×20 benchmark instances, there are 2002 operations in the alternative graph representation of each node of the B&B search tree. Since the GPU hardware limit is 1024 threads par block, we adapt the PEB and PESB approaches to enable each thread to handle two operations instead of one. This allows us to treat such huge instances.

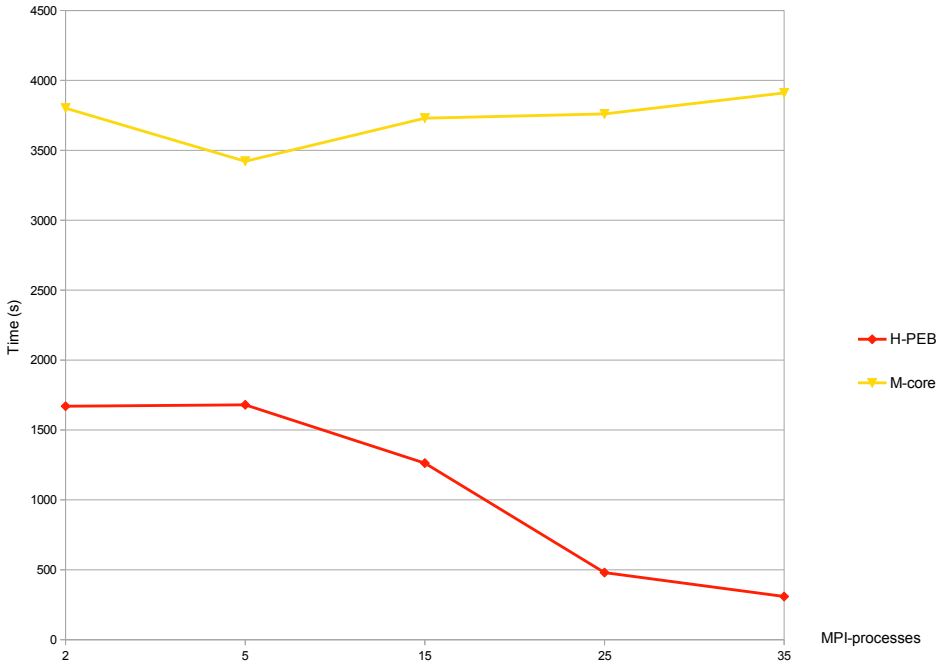


Figure 4.6.1: Impact of using different number of MPI-processes to explore 700,000 nodes for Tai61 instance.

Figure 4.6.1 shows the execution time needed for the Multi-core CPU and H-PEB approaches to explore 700,000 nodes using a different number of MPI-processes. For the Multi-core approach, the best time is reached for **five** MPI-processes. After that, we notice an increase in execution time when increasing the number of parallel MPI-processes. This can be explained by the limited number of CPU-cores available in our workstation which is four. Therefore, the worker's tasks are executed sequentially when the number of workers is above four. For the Hybrid H-PEB approach, the best time is reached for 35 MPI-processes which

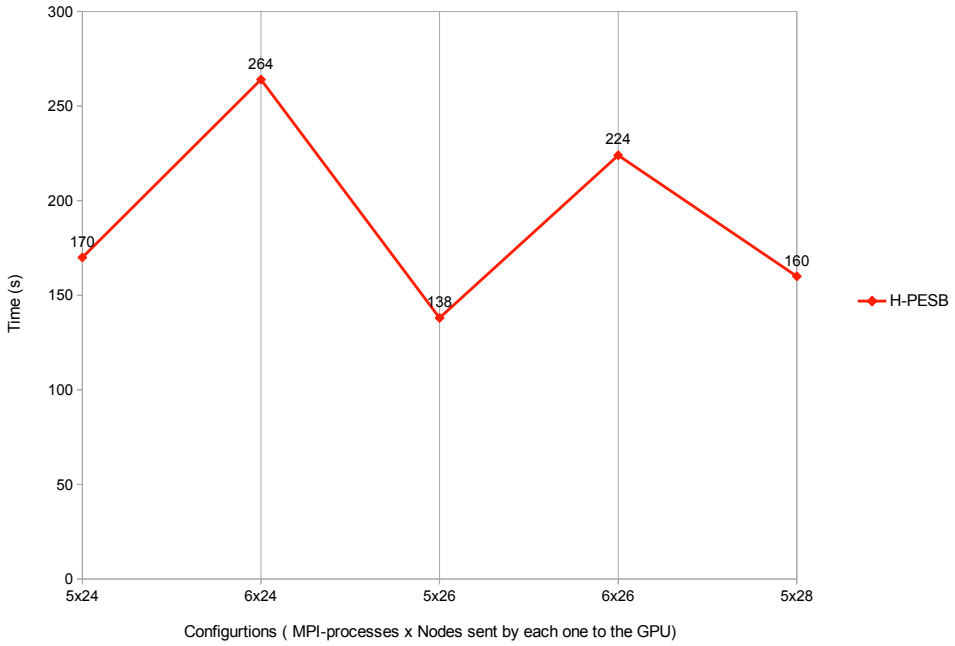


Figure 4.6.2: H-PESB execution time using different configurations to explore 700,000 nodes for Tai61 instance.

is the maximum supported by Nvidia MPS. This hybrid version supports a large number of workers compared to the Multi-core version since each worker has less than 10% of its execution time on the CPU. Hence, the four CPU-cores can handele a large number of parallel processes.

For the H-PESB approach, there is not an easy way to find the best configuration. For

Table 4.6.4: Average execution time (in seconds) of the proposed approaches to explore 700,000 nodes.
nb-pr: The number of parallel (host) processes running simultaneously.
nb-nodes: The number of nodes evaluated simultaneously on the GPU by each parallel process.

Size	$B\&B_{Seq.}$	$B\&B_{Mcore}$	*PEB* (nb_pr=1)		*H-PEB* (nb_pr=35)	
			nb-nodes	time	nb-nodes	time
15×15	188	52	1	603	1	162
20×15	384	113	1	653	1	164
20×20	393	120	1	736	1	173
30×15	1076	375	1	795	1	180
30×20	1127	447	1	955	1	209
50×15	4246	1454	1	1162	1	270
50×20	10546	3728	1	1530	1	340
100×20	69300	19200	1	3760	1	741

each instance, we have to test several configurations and take the best one among them. Each configuration is defined by the number of MPI-processes and the number of nodes sent to the GPU by each MPI-process at each iteration. As illustrated in Figure 4.6.2, the best performance for the H-PESB approach is reached for a low number of MPI-processes as compared to the H-PEB approach. This can be explained by the limited amount of virtual memory available in the Nvidia MPS server which influences the number of physical contexts that the latter can create (one context for each parallel process). Indeed, the larger the amount of virtual memory allocated to each process, the lower the number of parallel processes that the MPS server can mange. Hence, the larger the number of nodes sent to the GPU by each process, the limited the number of parallel processes that the MPS server can manage. This is why the H-PEB approach can simultaneously manage a large number of physical context (up to 35) since each process uses a low virtual memory matching the size of only one node. We also notice from Figure 4.6.2 that it is more beneficial for the H-PESB to increase the number of nodes sent to the GPU instead of increasing the number of MPI-processes. For that, we opted for the experiments to fix the number of MPI-processes to five and to increase the number of nodes sent to the GPU while it reduces the execution time needed to explore 700,000 nodes.

Table 4.6.4 and Table 4.6.5 report the average execution times for each approach to explore 700,000 nodes. The first column (*Size*) reports the size of the benchmark instances. Column $B\&B_{Seq}$ reports the average execution time of the optimized serial B&B algorithm. Column

Table 4.6.5: Average execution time (in seconds) of the proposed approaches to explore 700,000 nodes.
nb-pr: The number of parallel (host) processes running simultaneously.
nb-nodes: The number of nodes evaluated simultaneously on the GPU by each parallel process.

Size	$B\&B_{Seq.}$	$B\&B_{Mcore}$	*PESB* (nb_pr=1)		*H-PESB* (nb_pr=5)	
			nb-nodes	time	nb-nodes	Time
15×15	188	52	240	37	90	60
20×15	384	113	240	53	90	59
20×20	393	120	240	71	86	58
30×15	1076	375	240	104	54	60
30×20	1127	447	140	156	46	71
50×15	4246	1454	80	280	34	103
50×20	10546	3728	30	396	26	145
100×20	69300	19200	20	1050	20	418

$B\&B_{Mcore}$ gives the execution time obtained by our Master/worker approach exploiting only the Multi-cores CPU using 4 workers. For all other approaches, columns *Time* and *nb-nodes* report respectively the average execution time needed by each approach to explore 700,000 nodes and the number of nodes sent to the GPU at each iteration. For each column the parameter *nb_pr* indicates the number of parallel processes running simultaneously on the CPU according to the master/worker paradigm. Column *PEB* reports the results of our GPU node-based approach obtained by sending one node at a time for parallel evaluation on the GPU. Column *PESB* reports the results of our second GPU based approach obtained by sending several nodes to the GPU to be evaluated simultaneously using several GPU thread blocks. Columns *H-PEB* and *H-PESB* reports the results of the hybridization between the Multi-core approach and *PEB, PESB* GPU approaches using Nvidia MPS. *i.e.* both master and workers use simultaneously the GPU to accelerate respectively their bounding processes.

As mentioned before, 35 MPI-processes are used in the *H-PEB* approach and each one of them uses the default CUDA stream to launch its kernel to evaluate one node at a time. For the *H-PESB* approach, we fixed the number of parallel processes to five due to the huge amount of virtual memory matching the number of nodes sent to the GPU.

We notice from Table 4.6.4 that the complexity and the execution time increase when increasing the size of instances. Therefore, the need for parallelization is crucial.

Figure 4.6.3 shows the histogram representation of the execution time for the different

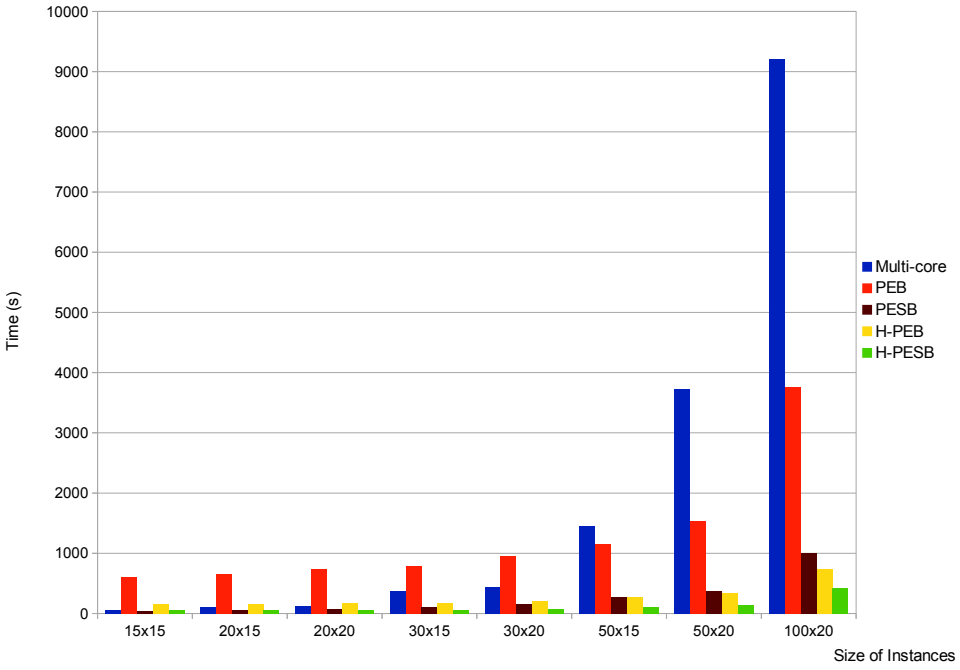


Figure 4.6.3: Execution time of the proposed approaches.

approaches. The first result from Table 4.6.4 and Figure 4.6.3 is the positive impact of using parallel architectures to reduce the execution time needed to solve the BJSS problem. The improvement obtained with the Multi-core version is limited which is expected since our workstation contains only four CPU-cores. Therefore, increasing the number of workers above four reduces further the obtained performances.

For the PEB version, we notice a low performance for small instances (15×15, ..., 30×20) against the Multi-core and sequential approaches. This can be explained by the high ratio of communication to computation on the GPU. i.e. The approach passes the most of the time in sending data and recovering results to/from the GPU. By increasing the size of instances, we notice a significant improvement in execution time as compared with the sequential and multi-core cases. In addition to its efficiency in reducing the execution time for large instances, the PEB approach does not depend on the GPU capacity since it uses a small amount of the GPU resources. The results from Table 4.6.4 show that our PEB approach is up to 18 times faster as compared with the serial B&B version and up to 5 times faster as compared to the multi-core version. However, only one thread-block is used in this approach and the others remain idle.

The performance of the *PESB* approach depends on the number of nodes that the GPU can evaluate simultaneously which is determined essentially by the amount of the shared memory used by each GPU block to evaluate a node. The number of nodes evaluated simultaneously for small instances is equal to the maximum number of blocks that our GPU can run simultaneously which is 240. By increasing the size of instances, the number of nodes evaluated simultaneously, matching the number of parallel blocks that a GPU can handle, decreases. This behavior can be explained by the increase in the amount of shared memory needed by each block to handle the evaluation of a node. Since the amount of shared memory is fixed, the number of parallel block decreases by increasing the shared memory needed by each block. By increasing the ratio of computation to communication on the GPU, we have been able to reduce further the execution time. Indeed, the *PESB* approach is three times faster compared to the *PEB* approach, 18 times faster compared to the multi-core version, and 65 times faster as compared with the optimized serial B&B version.

The hybrid *H-PEB* approach considerably reduces the execution time even for small instances against the sequential approach. This performance represents the results of exploiting both the CPU-cores and the GPU at the same time by using concurrent kernels execution provided by Nvidia MPS. This tool allows us to increase the GPU occupation. Furthermore,

the wasted time in CPU/GPU communications is covered by the concurrent access to the GPU where several workers execute their bounding operation at the same time. Unlike for small instances, the *H-PEB* approach outperforms the results of the *PESB* approach for large instances. This can be explained by the limited number of nodes that the PESB approach can handle simultaneously for large instances. This hybridization allowed us to reduce the execution time by a factor of 5x as compared with the PEB approach and a factor of 26x as compared with the multi-core version. Therefore, reaching a speedup around 94x.

Our last parallel approach (*H-PESB*) is also based on the Nvidia MPS. It represents a hybridization between the multi-core and the *PESB* approaches. This approach fully occupies the GPU which explains its good performance even for small instances. The results of this approach show a relative speedup up to 160. The results suggest that the H-PESB approach is two times faster as compared with the *PESB* approach and 46 times faster as compared with the multi-core version.

The results of our two hybrid approaches show clearly the benefit of combining both high level and low-level parallelization using Nvidia MPS. This tool allows to take benefit of both the multi-core CPU and the GPU at the same time. Thus, we can have several processes executing instructions on the GPU while others exploiting the multi-core CPU to perform branching and elimination operations.

We notice from our experience of using Nvidia MPS that the best performance is obtained for a large number of parallel host processes. This fact can be noticed from the H-PEB results in table 4.6.4 and Figure 4.6.1. However, the number of parallel processes depends on the MPS server resources. More precisely, the amount of virtual memory space available in the MPS server. This explains why we couldn't use a large number of host processes in the H-PESB approach since the virtual memory allocated to each parallel process is significant, matching the pull of nodes sent for evaluation on the GPU. Table 4.6.6 shows the number of

Table 4.6.6: The number of GPU communications needed for each approach to explore 700,000 nodes.

Approaches	#processes	#nodes sent	GPU communications
Multi-core	5	0	0
PEB	1	1	1400,000
H-PEB	35	1	40,000
H-PESB	5	20	14,000

GPU communications needed for each approach to explore 700,000 nodes. For each approach, column *processes* reports the number of used parallel processes. Column *nodes sent* reports the number of nodes sent by each process. Finally, column *GPU-communications* reports the number communications between the CPU and the GPU. The PEB approach has a huge number of CPU-GPU communications since two communications are needed to evaluate one node at each iteration. For the H-PEB approach, we have 40,000 communications since this latter supports up to 35 connections at the same time to the GPU without blocking. The same for the H-PESB approach, the latter reduces more the number of communications to the GPU which explains the obtained performances.

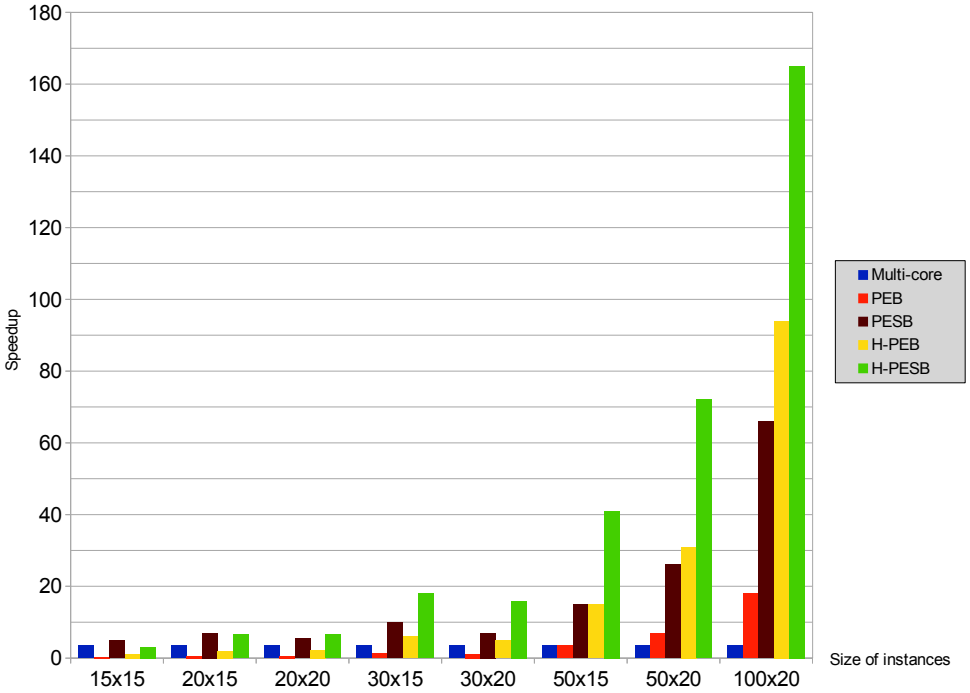


Figure 4.6.4: The speedup of the proposed approaches.

Figure 4.6.4 shows the relative speedups of our proposed approaches for different benchmark sizes. The speedup of our Multi-core version is around four for all sizes which is expected since it depends on the number of CPU-cores available in our workstation. The speed-up of the other approaches is proportional to the size of instances. Therefore, the maximum speedup is obtained for the 100×20 instances. This is logical because the speedup of these

approaches depends on the amount of the computation on the GPU. The idea used in the PEB approach to accelerate the bounding of one node at a time on the GPU using several threads organized into one block showed a speedup around 18x as compared to the serial version. The *PESB* approach gave the best performance against all other approaches for small instances. However, this is not the case for large instances as compared with the H-PEB and *H-PESB* approaches due to the limited number of nodes evaluated simultaneously on the GPU. This can be explained by the limited amount of shared memory available in the device and the large amount of this memory needed by each GPU block for synchronization. The speedup obtained by the hybrid H-PEB approach is around 90 times faster which confirms the efficiency and the benefit of using both CPU-cores and GPU at the same time. The *H-PESB* approach has achieved the best performances for almost all sizes against all other approaches. It has achieved an impressive speedup, especially for the largest instances where it is up to 160 times faster than a sequential B&B algorithm. In addition, the speedup of hybrid approaches grows according to the size of instances. This is due to the ratio of computation to communication that increases by increasing the size of instances. This proves that the hybrid approaches are scalable and can easily deal with large instances. These last two approaches are based on the concurrent kernels execution via Nvidia Multi Processes Service (MPS) which is rarely exploited in scientific computing.

The performance (speedup) of the hybrid approaches is the result of: (1) The use of the PEB scheme, which is 18 times faster, as a base of our hybrid approaches. (2) The Exploitation of both the power of the CPU-cores and the GPU at the same time using Nvidia MPS. (3) The occupation of the GPU over time *i.e.* several workers run instructions on the GPU while others perform data-transfer from/to the GPU and yet others apply elimination and branching operations on the CPU.

4.6.3 Results of the Tree-based Parallelization Approach

In the following, we report the impact of exploring the search tree in parallel on both the execution time and the solution quality. The experiments have been carried out using a cluster configuration which is a set of computing nodes interconnected by a high-speed communication network. Each node has a bi-processor and each one has eight cores with 2 GHz clock speed. The approach has been implemented using C++ language and Message Passing Interface (MPI) [44] as a communication tool between processes. In order to measure the efficiency of our proposed parallel approach named $B\&B_{NS}$, we first compare our optimal

Table 4.6.7: Our $B\&B_{NS}$ makespan results Vs. the literature B&B method for the BWS problem.

Instance	*Size*	$B\&B_{NS}$					$B\&B_{MP}$
		UB	$C_{optimal}$	$T_{seq.}$	$T_{par.}$	*Speedup*	[54]
La01	10×5	820	**793**	4.2	1.0	4	-
La02	10×5	793	**793**	18.9	1.0	18	-
La03	10×5	740	**715**	5.5	1.0	5	-
La04	10×5	764	**743**	5.6	1.0	5	-
La05	10×5	666	**664**	6.9	1.0	6	-
La16	10×10	1142	**1060**	551.8	9.1	62	1060
La17	10×10	977	**929**	314.6	13.3	26	929
La18	10×10	1078	**1025**	1160.6	8.1	145	1025
La19	10×10	1093	**1043**	1773.5	26.1	70	1043
La20	10×10	1154	**1060**	162.2	10.5	16	1060
La06	15×5	1180	**1060**	132383.2	1179.8	112	-
La07	15×5	1084	**1016**	265621.3	3676.4	72	-
La08	15×5	1125	**1040**	109313.5	1089.4	100	-
La09	15×5	1223	**1141**	133252.7	3288.2	41	-
La10	15×5	1203	**1096**	102632.6	530.9	194	-

results with the state of the art results. Among the works on solving optimally the BJSS problem we find: the B&B method ($B\&B_{MP}$) proposed by Mascis and Pacciarelli [54] and the one proposed by AitZai et al. [2]. To the best of our knowledge these results represent the only works on solving optimally the BJSS problem till now.

Table 4.6.7 summarizes our obtained results. The first two columns report the instance name and size. For our parallel B&B method named $B\&B_{NS}$, Column *UB* reports the upper bound used to compute the optimal solution. Column $C_{optimal}$ reports the *Makespan* (Cmax) of the optimal solution found by our approach. Column $T_{seq.}$ reports the time needed by our sequential version to find the optimal solution and complete the exploration of the search tree. Similarly, Column ($T_{par.}$) reports the time needed by our high level parallel B&B (cluster-based), using 128 CPU-cores, to find the optimal solution and complete the exploration of the search tree. Column *speedup* is defined as the ratio between the sequential and parallel execution times. Finally, column $B\&B_{MP}$ shows the *Makespan* of the instances that Mascis and Pacciarelli [54] solved optimally.

The most important results from Table 4.6.7 are the new optimal solutions found first by our parallel approach. Indeed, we have been able to solve optimally ten benchmark instances

(La01,..., La10) in which a proven optimum was unknown before.

It is noted that the B&B method proposed in [2] reported results for small local instances only, therefore, we could not make a comparison. Both of the $B\&B_{MP}$ and $B\&B_{NS}$ methods reached the optimal solutions for the 10×10 benchmark instances (La16,...,La20). Preliminary results indicate the efficiency of our exploration and selection strategy. Concerning the new optimal solutions found by our $B\&B_{NS}$ we can notice two sets. The first benchmarks set (La01–La05) is relatively easy to solve and takes only a few seconds while the second benchmarks set (La05,..., La10) is more difficult to solve and takes between 3 and 7 days of running time. Columns $T_{par.}$ and *Speedup* show also a large decrease of execution time and good speedup which reaches 194 times faster by using only 128 CPU-cores. The results also confirm the efficiency of our parallel approach.

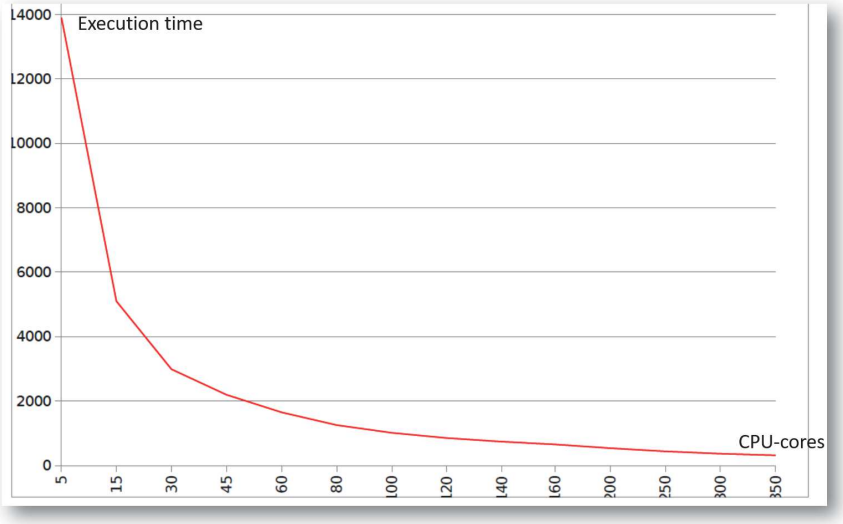


Figure 4.6.5: Execution times for solving la10 instance using different number of cores.

Moreover, figures 4.6.5 and 4.6.6 show respectively the variation of the execution time and efficiency of our parallel $B\&B_{NS}$ in solving the La10 benchmark using different number of CPU-cores (from 5 to 350 cores). The efficiency is calculated as the ratio between the speedup and the numbers of used CPU-cores. In Figures 4.6.5 and 4.6.6 we notice the existence of two phases: The first phase in which the number of cores is between 5 and 120, is characterized by efficiency greater than 1. This indicates a super-linear acceleration which explains the rapid decrease of the execution time in this phase. The second phase between 120 and 350

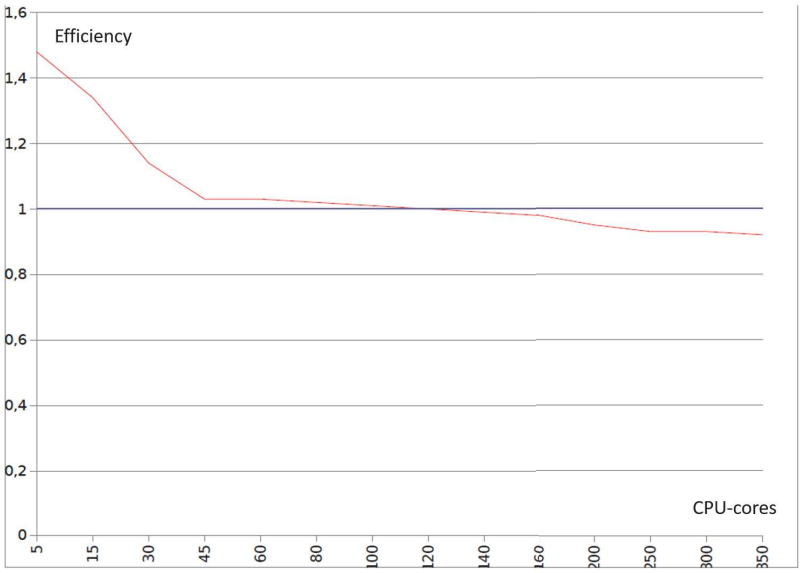


Figure 4.6.6: Measurement of the efficiency for solving La10 instance using different number of cores.

cores is characterized by a slow decrease of execution time and an efficiency lower than 1. This can be explained by the increase in the ratio of communication to computation. The super-linear acceleration can be explained by the rapid improvement of the UB which allows to avoid the exploration of several branches explored in the sequential B&B search tree.

Table 4.6.8: $B\&B_{NS}$ Efficiency measures for large BJSS instances.

Size	*Seq.* $B\&B_{NS}$	*Par.* $B\&B_{NS}$ 160 cores	*Speedup*	*Eff.*
20×5	10810141	1401436755	130	0.81
15×10	12586236	1425808495	113	0.70
20×10	4974214	608953176	122	0.77
30×10	1858076	195486802	105	0.66
15×15	10075595	1321763137	131	0.82
	Avg.		120	0.76

In the following, we evaluate the ability of the proposed parallel $B\&B_{NS}$ to accelerate the execution time for the large benchmark instances. Table 4.6.8 summarizes our obtained

Table 4.6.9: $B\&B_{NS}$ makespan results Vs. the best known solutions for the BWS problem.

Instance	*Size*	*Reference*	*Best*	*UB*	$B\&B_{NS}$	
					$C_{sequential}$	$C_{parallel}$
La01	10 × 5	IFS [61], IG [63]	793	820	**793**	**793**
La02	10 × 5	IFS [61], IG [63]	793	793	**793**	**793**
La03	10 × 5	IFS [61], IG [63]	715	740	**715**	**715**
La04	10 × 5	IFS [61], IG [63]	743	764	**743**	**743**
La05	10 × 5	IFS [61], IG [63]	664	666	**664**	**664**
La06	15 × 5	IFS [61]	1064	1180	-	**1060**
La07	15 × 5	IG [63]	1020	1084	-	**1016**
La08	15 × 5	IFS [61]	1062	1125	-	**1040**
La09	15 × 5	IG [63]	1162	1223	-	**1141**
La10	15 × 5	IFS [61]	1110	1203	-	**1096**
La11	20 × 5	IFS [61]	1466	1584	-	**1465**
La12	20 × 5	IG [63]	1271	1391	-	**1253**
La13	20 × 5	IFS [61]	1465	1541	-	**1383**
La14	20 × 5	IFS [61]	1506	1620	-	**1467**
La15	20 × 5	IFS [61], IG [63]	1517	1630	-	**1511**
La16	10 × 10	IG [63]	1060	1142	**1060**	**1060**
La17	10 × 10	IG [63]	929	977	**929**	**929**
La18	10 × 10	IG [63]	1025	1078	**1025**	**1025**
La19	10 × 10	IG [63]	1043	1093	**1043**	**1043**
La20	10 × 10	IG [63]	1060	1154	**1060**	**1060**
La21	15 × 10	IG [63]	1490	1554	-	**1433**
La22	15 × 10	IG [63]	1339	1458	-	1424
La23	15 × 10	IG [63]	1445	1570	-	**1436**
La24	15 × 10	IG [63]	1434	1546	-	**1354**
La25	15 × 10	IG [63]	1392	1499	-	**1313**
La26	20 × 10	IG [63]	1989	2125	-	1994
La27	20 × 10	IG [63]	2017	2175	-	**2005**
La28	20 × 10	IFS [61]	2027	2071	-	**2020**
La29	20 × 10	IG [63]	1846	1990	-	**1725**
La30	20 × 10	IG [63]	2049	2097	-	2055
La31	30 × 10	CP_OPT [61]	2921	3137	-	3028
La32	30 × 10	CP_OPT [61]	3237	3316	-	3368
La33	30 × 10	CP_OPT [61]	2844	3061	-	3037
La34	30 × 10	CP_OPT [61]	2848	3146	-	2961
La35	30 × 10	CP_OPT [61]	2923	3171	-	3065
La36	15 × 15	IG [63]	1755	1919	-	**1701**
La37	15 × 15	IG [63]	1870	2029	-	**1848**
La38	15 × 15	IFS [61]	1708	1828	-	**1598**
La39	15 × 15	IG [63]	1731	1882	-	**1714**
La40	15 × 15	IG [63]	1743	1925	-	**1714**

Table 4.6.10: $B\&B_{NS}$ makespan results Vs. the best known solutions for the BNS problem.

Instance	*Size*	*Reference*	*Best*	*UB*	$B\&B_{NS}$	
					$C_{sequential}$	$C_{parallel}$
La01	10 × 5	MX [55], IG [63]	881	1020	**881**	**881**
La02	10 × 5	MX [55], IG [63]	900	1063	**900**	**900**
La03	10 × 5	MX [55], IG [63]	808	1043	**808**	**808**
La04	10 × 5	MX [55], IG [63]	859	1064	**859**	**859**
La05	10 × 5	MX [55], IG [63]	732	936	**732**	**732**
La06	15 × 5	IG [63]	1225	1360	-	**1194**
La07	15 × 5	IG [63]	1133	1283	-	**1127**
La08	15 × 5	MX [55]	1216	1369	-	**1173**
La09	15 × 5	IG [63]	1311	1469	-	**1309**
La10	15 × 5	IG [63]	1237	1506	-	**1218**
La11	20 × 5	IG [63]	1641	1832	-	**1587**
La12	20 × 5	IG [63]	1465	1548	-	**1410**
La13	20 × 5	IG [63]	1627	1941	-	**1569**
La14	20 × 5	IG [63]	1686	1833	-	**1608**
La15	20 × 5	IG [63]	1680	1974	-	**1639**
La16	10 × 10	IG [63], MX [55]	1148	1369	**1148**	**1148**
La17	10 × 10	IG [63], MX [55]	968	1563	**968**	**968**
La18	10 × 10	IG [63], MX [55]	1077	1678	**1077**	**1077**
La19	10 × 10	IG [63], MX [55]	1102	1443	**1102**	**1102**
La20	10 × 10	IG [63], MX [55]	1118	1498	**1118**	**1118**
La21	15 × 10	IG [63]	1627	1954	-	**1539**
La22	15 × 10	IG [63]	1426	1658	-	**1391**
La23	15 × 10	IG [63]	1574	1750	-	**1548**
La24	15 × 10	IG [63]	1502	1646	-	**1469**
La25	15 × 10	IG [63]	1533	1799	-	**1451**
La26	20 × 10	IG [63]	2146	2425	-	**2118**
La27	20 × 10	IG [63]	2191	2575	-	**2179**
La28	20 × 10	IG [63]	2245	2371	-	**2203**
La29	20 × 10	IG [63]	2030	2390	-	**2026**
La30	20 × 10	IG [63]	2242	2497	-	**2185**
La31	30 × 10	IG [63]	3219	3537	-	3284
La32	30 × 10	IG [63]	3567	3716	-	3660
La33	30 × 10	IG [63]	3401	3061	-	**3199**
La34	30 × 10	IG [63]	3202	3546	-	3295
La35	30 × 10	MX [55]	3373	3671	-	3424
La36	15 × 15	IG [63]	1835	1919	-	**1814**
La37	15 × 15	IG [63]	1931	2029	-	1961
La38	15 × 15	IG [63]	1813	1828	-	**1752**
La39	15 × 15	IG [63]	1811	1882	-	**1776**
La40	15 × 15	IG [63]	1815	1925	-	**1806**

results. The first column reports the instance size. Column *Seq.* $B\&B_{NS}$ and *Par.* $B\&B_{NS}$ give respectively the average number of explored nodes of our sequential and parallel $B\&B_{NS}$ in one hour execution time. Column ***speedup*** reports the speedup obtained by using 160 CPU-cores. Finally, the efficiency *Eff.* is calculated as the ratio between the speedup and the number of used CPU-cores.

The obtained results indicate that for all instances, the parallel approach is much faster than the sequential approach. Using 160 CPU-cores, the parallel $B\&B_{NS}$ is 120 times faster than the sequential approach with an average efficiency of 0.76. This confirms the efficiency and the scalability of our proposed parallelization.

In the following, we will consider our parallel B&B as a metaheuristic. This can be achieved by considering the best explored solution during 7200 seconds as an approximate result of our parallel approach. In order to prove the ability of our proposition to improve the solution quality, even for large instances, we compare our approximate makespan results with the best solution to our knowledge. As mentioned earlier, the BJSS problem has two cases namely the BJSS with swap (BWS) and the BJSS with no swap (BNS). Among the most recent works on BJSS with swap we find: the Tabu Search method proposed by Gröflin et al. [42, 43], the *IFS* algorithm and the *CP-OPT* method presented by Oddi et al. in [61], and finally, the Iterated Greedy algorithm (*IG*) proposed by Pranzo and Pacciarelli in [63]. To the best of our knowledge, these results represent the best-known solutions.
Our results are obtained by lunching our parallel B&B on 320 CPU-cores during 7200 seconds (one execution only). This time represents the same execution time required for each instance by *IG* [63]. The reported results in *IFS* [61] required 2 runs of 1800 seconds for each instance and configuration. Since there are 16 configurations, the total computation time required by *IFS* for each instance is 16 hours.

Table 4.6.9 and Table 4.6.10 summarizes respectively the obtained results for the BWS and the BNS versions. The first two columns report the instance name and size. Column 3 gives the reference to the best-known solutions which are reported in column 4 (C_{Best}). Finally, column $C_{parallel}$ and $C_{sequential}$ report respectively the Makespan results found by our parallel and sequential B&B method over 7200 seconds. For each instance, the bold results indicate that our approach reached the best solution in the literature and bold underline results indicate that our approach improved the best solution in the literature.

Table 4.6.9 shows a good improvement of the solutions quality. More precisely, over 40 experimented benchmark instances, our parallel *B&B* results ($C_{parallel}$) improves the best-known

solutions for 17 instances. The results show also that our parallel approach matched the best result found in the literature for ten instances. Moreover, the deviation (Relative error) for the non-improved benchmark instances varies between 0.3% and 7%. Unlike our parallel *B&B* method, our sequential version fails to improve most benchmarks, especially for the large instances. This can be explained by the long time needed by the sequential B&B to reach solutions in the search tree.

In the following, we report the approximate results of our parallel B&B algorithm for the BJSS problem with no swap (BNS). Despite the large number of application areas for the BNS problem, this special case has been treated by few authors. Among the most recent works on solving the BNS problem we find: the IG metaheuristic proposed by Pranzo and Pacciarelli [63] and the TS algorithm proposed by Mati and Xie [55] referred to as MX. To the best of our knowledge, their results represent the best-known solutions for the BNS case.

The results reported in Table 4.6.10 confirms the results found earlier for the BWS case. Indeed, the approximate results of our proposed parallelization show a good improvement in the solution quality for a large number of instances. Over 40 experimented benchmark instances, our parallel *B&B* results ($C_{parallel}$) improves the best-known solutions for 25 benchmark instances and reach the best solution for ten benchmark instances. The obtained results in Table 4.6.9 and Table 4.6.10 validate our idea of using the parallel B&B algorithm as an approximate method to solve a complex scheduling problem.

4.7 Conclusions

This chapter investigates the implementation and the parallelization of the Branch and Bound algorithm using Multi and Many-core systems in order to solve optimally the NP-hard Blocking Job Shop Scheduling problem (BJSS). To this aim, five parallel approaches in addition to the serial one have been proposed.

In our serial implementation, we proposed a new selection and exploration strategies aiming to improve the efficiency of the B&B algorithm. The results where promising, however, the algorithm remains inefficient for large benchmark instances. Thus, the parallelization is unavoidable. With the aim of exploiting efficiently high-performance computing architectures, five parallel approaches have been proposed. The first approach, called Parallel evaluation of the Bound (PEB), is a GPU node based parallelization in which the bounding of one node at a time is calculated in parallel on the GPU. Experiments showed that this

approach is up to 18 times faster as compared to the sequential approach. However, small amount GPU resources are used in this latter. To increase the GPU utilization, a second parallel approach called Parallel Evaluation of Several Bounds (PESB) is proposed. This approach represents a generalization of the PEB approach obtained by sending several nodes to the GPU at each iteration instead of one node at a time. By increasing the GPU occupation, we have been able to increase the speedup to reach 66x. In order to fully occupy the GPU over time, hybridization between the high-level B&B version, exploiting the multi-core CPUs, and GPU based low-level approaches have been proposed. The proposed approaches here are based on Nvidia Multi-Processes Service (MPS). This tool allows several host processes to use simultaneously the GPU resources to execute their kernels. Two parallel approaches exploiting this feature have been proposed namely H-PEB and H-PESB. These approaches have several B&B instances exploring the search tree in parallel and each instance uses the GPU for the evaluation of one or several nodes at a time. The proposed approaches allow to increase the GPU occupation over time i.e. several host processes execute instructions on the GPU while other processes perform branching and elimination operations on the CPU. The obtained results confirm the efficiency of our proposals and the positive impact of increasing the GPU occupation over time by using concurrent kernels execution provided by Nvidia MPS. The results showed a relative speedup of 160x for large instances as compared with an optimized sequential B&B approach. The fifth parallel approach designed for cluster-based architectures exploits the fact that each search tree node can be explored independently of the others. In other words, the fifth approach can be seen as a high-level B&B parallelization in which hundreds of cores explore the B&B search tree simultaneously. Two goals are designed for this approach. The first goal is to accelerate the B&B execution time and solve complex benchmark instances, while the second goal is to use the same approach to report the approximate result for large benchmark instances. The experiments using hundreds of cores allowed us to solve optimally for the first time ten benchmark instances. In addition, the approximate results of our approach improved the best-known results for over 40 benchmark instances which validate our idea to use the B&B algorithm as an approximate approach.

As a conclusion, we have seen the positive impact of using parallel architectures to accelerate the B&B execution time. However, this method remains inefficient for large benchmark instances. For this reason, we propose in the next chapter the adaptation and the parallelization of the Tabu search metaheuristic which has proven its efficiency for a wide range of optimization problems.

5

Parallel Tabu Search Method for the BJSS Problem

5.1 Introduction

The blocking job shop scheduling problem (BJSS) is a version of the classical job shop scheduling with no intermediate buffer between machines. The BJSS is known to be NP-hard in the strong sense. As all optimization problems, the BJSS can be solved using either exact or approximate methods. As we already saw in Chapter 4, solving optimally this problem by the means of the B&B algorithm is inefficient for dealing with large benchmark instances due to their huge search space. The only choice for dealing with such instances is to use approximate methods. These methods consist to perform a tradeoff between the quality of the obtained solution and the explored search space, hence, the running time. Metaheuristics can be grouped into two categories: population-based metaheuristics and solution-based metaheuristics. From the stat of the art on solving the BJSS problem, we can notice that solution-based metaheuristics are the most suitable for complex scheduling

problems. The choice of this metaheuristic is mainly motivated by the good results of this metaheuristics for the classical JSSP version and from our experience that involves using several population-based and solution-based metaheuristics. In this chapter, we propose an efficient adaptation of the Tabu search (TS) algorithm for the BJSS problem, in addition to several parallelization schemes that exploit HPC architectures. The TS algorithm is one of the widely used metaheuristics for solving optimization problems due to its high adaptability. This method represents a higher level heuristic procedure designed to guide other heuristics to escape the trap of local optimality. It consists to explore partially the search space by moving from one solution to another one using a neighborhood function. As already seen in Chapter 2, the BJSS problem has two versions namely: the blocking with swap (BWS) and the blocking no-swap (BNS). For the remaining of this chapter, all the proposed approaches are for both BJSS cases.

Several authors tried to solve the BJSS problems using metaheuristics. However, their obtained results are of poor quality since the classical reproduction techniques do not take into account the specification of the blocking constraint, which leads to explore massively unfeasible solutions. Indeed, applying the classical TS neighborhood to the BJSS problem produces infeasible solutions in most cases leading to a useless exploration of the search space, thus low results quality. To overcome this drawback, we propose in this chapter an efficient adaptation of TS algorithm to the blocking constraint by proposing a new neighborhood function based on the reconstruction strategy. This neighborhood consists to remove arcs causing the infeasibility and rebuild the neighbor solutions by using heuristics. Our goal from this neighborhood is the following: (1) to explore only feasible solutions and (2) to guide the search process using heuristics that allow to avoid a random exploration of the search space. Experiments on the reference benchmark instances show that our TS algorithm using the proposed neighborhood improves most of the best-known results in the literature and gives new upper bounds for more than 52 instances in both BJSS cases.

However, using heuristics to recover the feasibility of the neighboring solutions is a time-consuming operation which makes the TS algorithm very slow, since it spends a huge time to explore a small area in the search space. To overcome this drawback, we propose in the second part of this chapter several parallelization approaches for our proposed TS algorithm. The parallelization allows on one hand to accelerate the search process of the TS method and on the other hand, to diversify the search. The TS algorithm is very well adapted for parallelization which gives us the advantage of exploring simultaneously several areas in the

search space. Experiments using a cluster-based architecture with 240 CPU-cores show the positive effect of our parallel approaches combined with the proposed recovery strategy on the obtained solution quality. The results clearly indicate the significant impact of using parallel architectures to expand the explored search space. i.e. With the same execution time as the sequential version, the parallel approaches explore 240 times the search space explored by the sequential version which may lead to improve the solution's quality.

The remainder of this chapter is organized as follows: Section 2 describes the sequential TS algorithm and the proposed neighborhood function. Section 3 describes our proposed parallelization approaches that exploit HPC architectures. Computational results and discussions are reported in section 5. Finally, conclusions are presented in Section 5.

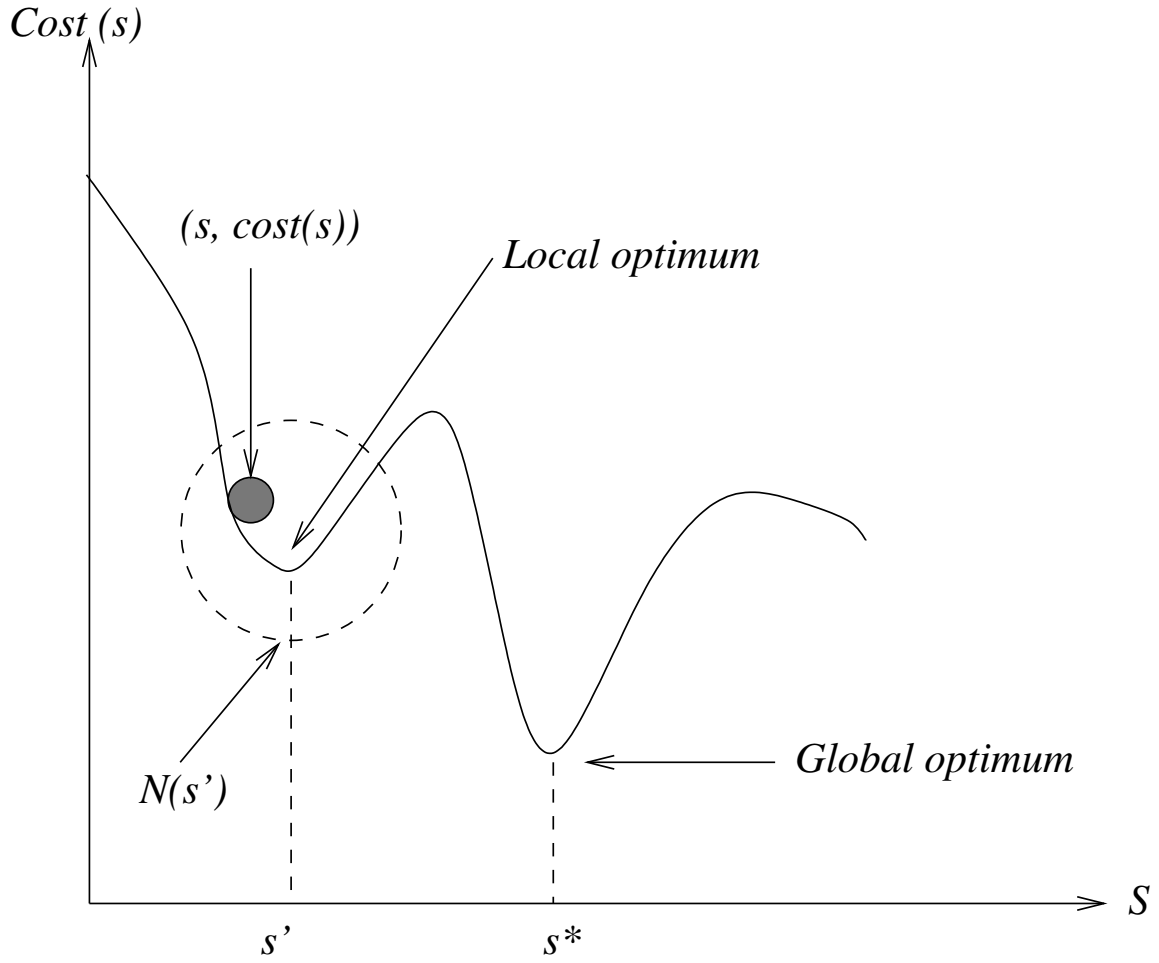


Figure 5.1.1: Local and global optimums.

5.2 Proposed Sequential Tabu Search Algorithm

The Tabu Search is a local search metaheuristic used to solve combinatorial optimization problems. It was introduced by Glover in 1986 [37]. At each iteration, the TS moves from one solution to another by exploring for a solution $s \in S$ the entire neighborhood $N(s)$ and the best solution in this neighborhood s' is selected as the new solution even if its quality is worse than s. To avoid the trap of local optimums, the latest k visited configurations are stored in a short-term memory forbidding any move that results in any of these configurations. This memory is called the *Tabu List* (TL). Moreover, Figure 5.1.1 describes the TS trajectory in the search space.

Sometimes, a good move that improves the best result already found by the algorithm is forbidden by the TL. In this case, the *aspiration criteria* allows us to overcome the tabu status. Finally, a *long term memory* is used to diversify the search by exploring new regions in the search space. The TS algorithm has been applied to most scheduling problem including the BJSS problem as in [42, 43]. For more additional information on the method, the reader may refer to Glover [37, 38].

In this section, we describe the proposed TS method with a new neighborhood function For both BJSS cases (BWS and BNS). The proposed neighborhood is based on the reconstruction strategy which allows us to guide the search process and recover the feasibility of neighboring solutions. The recovering process consists to remove arcs causing the infeasibility and use heuristics to reconstruct the new neighbor solutions.

Table 5.2.1: The description of the symbols used in our TS.

Symbol	Description
s	a feasible schedule (solution) for the BJSS problem.
S	a set of all feasible solutions for the BJSS problem.
$s*$	The best solution found by TS.
$Cost(s)$	The Cost (*makespan*) of the solution s.
sc	Best solution in Cand(s).
TL	Tabu List.
$Cand(s)$	a set of unforbidden neighbor solutions of the schedule s.
$N(s)$	a set of all neighbor solutions of schedule s.

Algorithm 2 and Table 5.2.1 describe the general structure and symbols used in the pro-

posed Tabu search algorithm.

Algorithm 2 Pseudo-code of the proposed Tabu Search algorithm (TS_{IN})

BEGIN
1. Generate an initial solution $s \in S$;
2. $s^* := s$;
3. $TL := \emptyset$;

REPEAT
4. $cand(s) := \{s' \in N(s) |$ the move from s to s' is not taboo OR s' satisfies the aspiration criterion $\}$;
5. Generate a solution $s_c \in Cand(s)$;
6. Update TL (insert the move from s to s_c in TL.);
7. $s := s_c$;
8. IF $cost(s) < cost(s^*)$ THEN $s^* := s$;

UNTIL stop-criteria = true
RETURN s^*;
END.

5.2.1 Initial Solution

The initial solution represents the starting point of the TS algorithm. This solution is chosen randomly and it is generally of poor quality. However, the TS algorithm is able to reach a good solution regardless of the quality of the initial solution. A very simple way to generate the initial solution, in both BJSS cases, is to schedule the jobs sequentially in a random way as explained in [63]. For example, a BJSS instance with three jobs we can have the following combinations *J1 J2 J3, J1 J3 J2, J2 J1 J3, J2 J3 J1, J3 J1 J2, and J3 J2 J1.* In this way, we are able to generate quickly an initial solution for our algorithm.

5.2.2 Proposed Neighborhood Structure

In this section, we present our proposed neighborhood function. For a better explanation, we use the notation of the alternative graph model [54]. The results of the TS method depend essentially on the quality of the used neighborhood function. Therefore, it is essential to be efficient and well adapted to the problem. Our proposed neighborhood function is based on the classical JSSP neighborhood (*N1*), where the neighbor solutions are obtained by

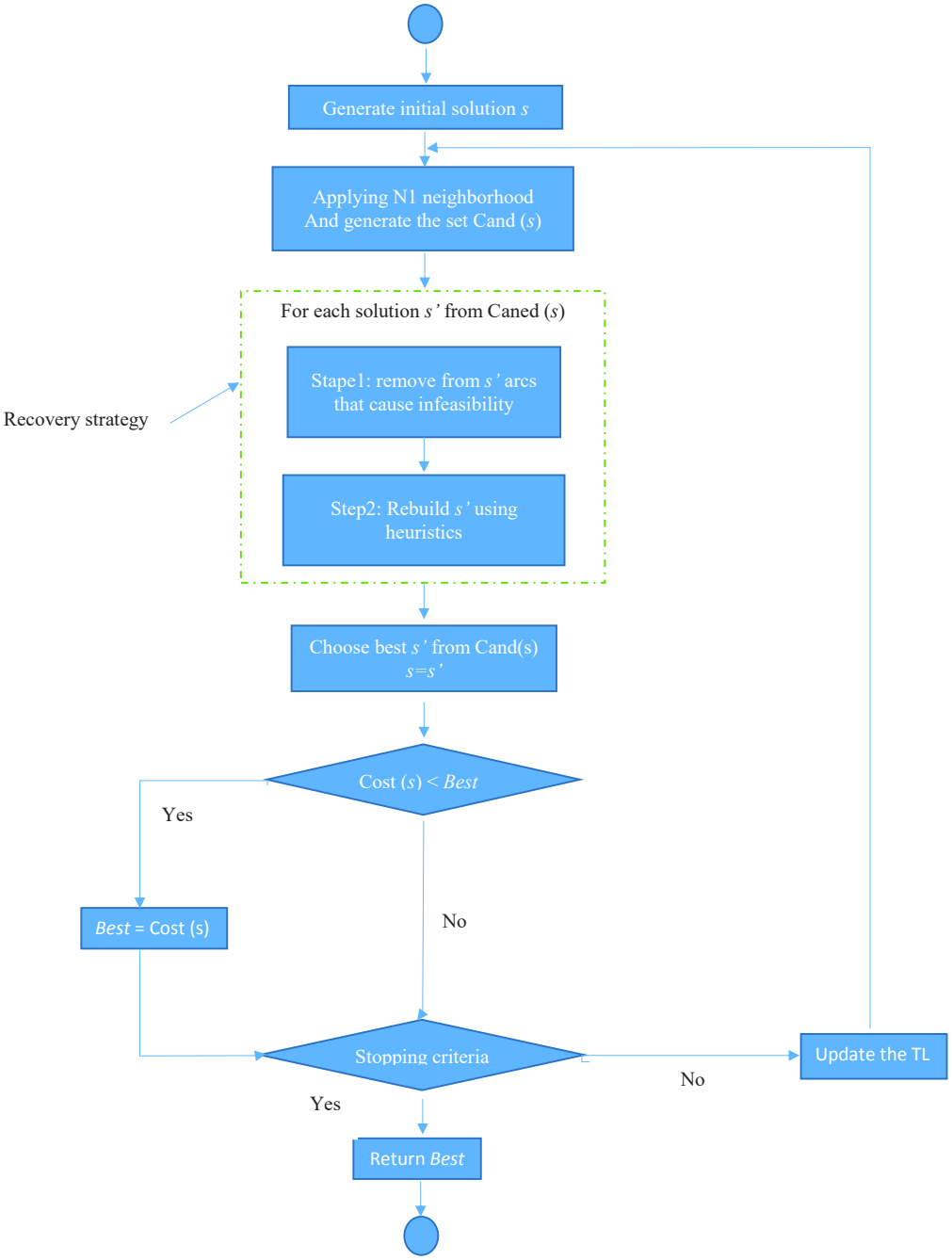


Figure 5.2.1: TS algorithm with the proposed neighborhood steps.

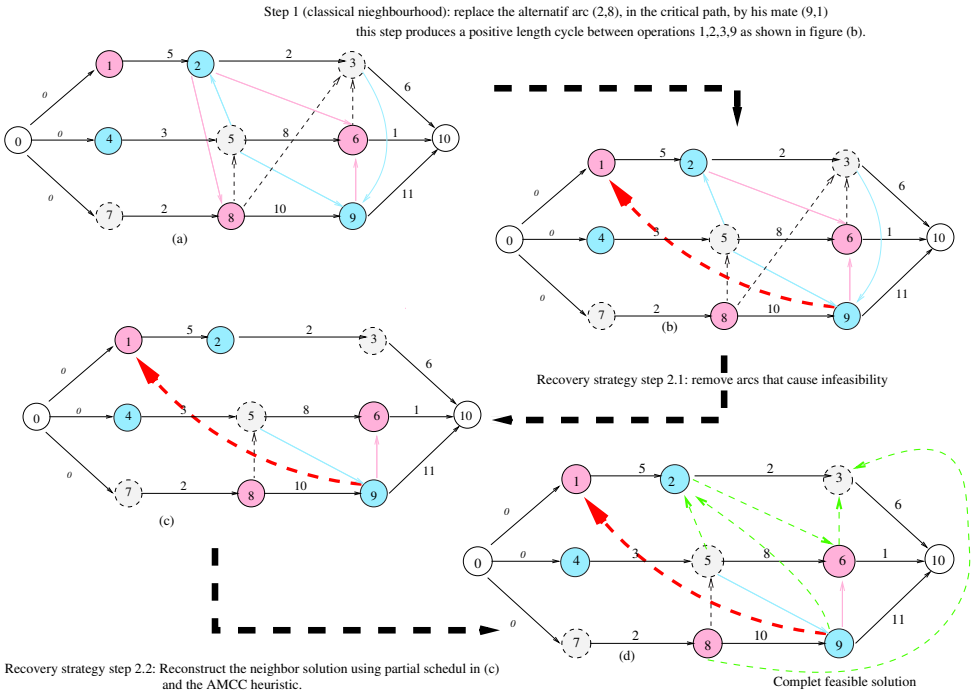


Figure 5.2.2: The proposed neighborhood steps.

permuting two successive critical operations on the same machine. In the classical JSSP, the N1 neighborhood produces always feasible solutions and converges to the local optimum [10]. However, applying the *N1* neighborhood to the BJSS problem produces in 98% of cases infeasible solutions. For this reason, Groflin and Klinkert [42, 43] propose a way to recover the feasibility of the neighbor solutions. However, the neighboring solutions are of poor quality.

Our Idea can be summarized in two steps. The first step consists to perform the N1 neighborhood, while the second step consists to recover the feasibility of neighboring solutions. This can be done by removing some arcs, related to *N1* move, causing the infeasibility and calling heuristics to complete the neighbor solutions as shown in Figure 5.2.1. For better understanding, Figure 5.2.2 shows an example of the proposed neighborhood steps.

Step 1: N1 Neighborhood

With N1, a neighbor solution s' of the solution s is obtained by permuting two successive critical operations o_i and o_j assigned to the same machine. Let us consider:

$$i' = \begin{cases} \sigma(i) & \text{if } o_i \text{ is a blocking operation.} \\ i & \text{if } o_i \text{ is an ideal operation.} \end{cases}$$

$$j' = \begin{cases} \sigma(j) & \text{if } o_j \text{ is a blocking operation.} \\ j & \text{if } o_j \text{ is an ideal operation.} \end{cases}$$

In the alternative graph representation, the neighbor solution (s') is obtained from s by replacing the alternative arc (i', j) on a critical path in $G(s)$ by its mate (alternative) (j', i). In 98% of the cases the neighbor solution s' is not feasible due to the existence of a positive length cycle.

Step 2: Recover the feasibility of the neighbor solution

This step consists to restore the feasibility of the neighbor solution (s') from the previous step by removing from s' some of the arcs that cause the infeasibility (Step 2.1) and then call the AMCC heuristic to complete the neighbor solution (Step 2.2). In the alternative graph, s' represents a complete selection where all alternative pairs are fixed; therefore, $A = \emptyset$ and $G(s') = (N, F \cup s')$ is the corresponding graph. Let us consider $J(i)$ as the job containing operation i (o_i).

Algorithm 3 Pseudo-code step 2.1

```
INPUT: complete selection s' and alternative arc (j', i).
OUTPUT: feasible partial selection s' and a set of unselected alternative pairs in A.
BEGIN
∀(l, m) ∈ s' − {(j', i)}
   BEGIN
      IF J(l) = J(i) OR J(m) = J(i)
      THEN
       BEGIN
         s' := s' − (l, m)
         A := A ∪ (l, m), (e,f).
         /* (e,f) is the alternative of arc (l, m) */
       END
   END
END.
```

Step 2.1: Removing arcs that cause infeasibility

After the replacement of the arc (i', j) by its mate arc (j', i) in step 1, we remove from s' all the incoming and outgoing alternative arcs from the job $J(i)$. The result of this step is a feasible partial selection s' and a set of unselected alternative pairs stored in A. In a more

Algorithm 4 Pseudo-code of the step 2.2 construction of the neighbor solution

INPUT: Alternative graph $G = (N; F; A)$
Feasible partial selection s'
OUTPUT: Solution (complete selection) s'
BEGIN
REPEAT
1. Select a pair $((l,m),(e,f)) \in A$ using AMCC ;
2. Select (l,m) *i.e.* $s' := s' \cup (l,m)$
3. $A := A - \{((l,m),(e,f))\}$;
4. IF $\exists((u,v),(p,q)) \in A : l(v,u) + a_{uv} > o$ and
$l(q,p) + a_{pq} > o$
5. THEN STOP, failed to find a feasible solution.
6. ELSE WHILE $(\exists((u,v),(p,q)) \in A : l(v,u) + a_{uv} > o)$;
BEGIN
7. $s' := s' \cup \{(p,q)\}$;
8. $A := A - \{(u,v),(p,q)\}$;
END
UNTIL $A = \emptyset$
RETURN s';
END.

formal way, Algorithm 3 describes this step using alternative graph model.

Step 2.2: construction of the neighbor solution

Given a partial selection s' from the Step 2.1, Algorithm 4 extends s' at each iteration with a new alternative arc (l,m) until a new feasible solution is obtained. To decide which alternative pair to choose first and which arc to select in order to feasibly complete s', we use the Avoid Maximum Current Cmax (AMCC) heuristic defined by Mascis and Pacciarelli [54]. This heuristic consists to avoid the unselected alternative arc that increases the most the *Makespan*. i.e., the AMCC heuristic selects the pair $((l,m),(e,f))$ such that: $l(o,e) + a_{ef} + l(f,n) = Max\{ l(o,u) + a_{uv} + l(v,n)\}\ \forall(u,v) \in A$. Therefore $s' := s' \cup (l,m)$ and $A := A - \{(l,m),(e,f)\}$. In order to avoid positive length cycles, Algorithm 4 (line 6) checks for each unselected alternative pair $((u,v),(p,q)) \in A$ the following:

1- IF $l(v,u) + a_{uv} > o$ THEN $s' := s' \cup (p,q)$ and $A := A - ((u,v),(p,q))$.

2- IF $l(q,p) + a_{pq} > o$ THEN $s' := s' \cup (u,v)$ and $A := A - ((u,v),(p,q))$.

This means that adding the arc (u,v) (resp. (p,q)) to s' will produce an infeasible selection, consequently the arc (p,q) (resp. (u,v)) is added to s'. If both alternative arcs (u,v) and (p,q) produce an infeasible selection (Algorithm 4, line 5) in this case, we say that the heuristic fails

to find a complete feasible selection which is very rare. In our experimentation, the AMCC heuristic has never failed to complete a neighbor solution. However, this may happen, in this case, the neighbor solution s' is rejected. At the end of this step, we have a complete feasible neighborhood solution s'. After the selection of each alternative arc by the algorithm, we systematically adjust the head and tail values for each operation in the graph. These values are used by the AMCC heuristic in order to choose the unselected alternative arc. The computation of the value $l(v,u)$ is basically calculated using the alternative graph but, we avoid the calculation if we can conclude directly that no path exists from operation v to operation u.

5.2.3 Tabu List

In order to avoid the trap of local optimum, a Tabu List (TL) is used. The TL elements must include enough information to faithfully remember the visited solutions. The size k of the TL depends on the benchmark size. In our case and after several experiments, the size of TL is is between ten and thirty items. The TL is updated at each iteration and First In First Out (FIFO) strategy is applied to the list. Given a candidate arc (i,j) and its alternative pair $((i,j),(h,k))$, the associated move consists to replace the candidate arc (i,j) by its alternative arc (h,k). To avoid returning back to the previous solution and going through the same solution a second we save both directions of the pair $((i,j),(h,k))$ associated with the performed move. In order to simplify the management of the TL, we propose the following codification of the alternative pair associated with the candidate arc to reverse (i,j).

$$\begin{cases} J_{min} = min\{J(i), J(j)\} \\ J_{max} = max\{J(i), J(j)\} \\ cod_{(i,j)} = ((M(j) * 10) + J_{min}) * 10 + J_{max}. \end{cases}$$

In this way, if we want to check whether a move is forbidden or not, we search for the codification of this move in the TL.

5.2.4 Aspiration Criterion

The aspiration criteria used in our case is the one used in most articles. It consists to perform a move that improves the best result already found by TS even if it is forbidden by the TL.

5.2.5 Long-Term Memory

In order to diversify the search, a *long term memory* is used. In most cases, the best solution in the neighborhood is selected for exploration and the second best one is never explored. For this reason, we use the *intermediate memory* to store the second best solution encountered during the search. When the search becomes stuck (no improvement of the solution quality) and after a fixed number of iterations, we initialize the TS with the solution stored in the *intermediate memory*.

5.3 Parallel Tabu Search Algorithm

The tabu search algorithm is suitable for parallelization since it can be easily adapted to benefit from parallel computing architectures in order to explore efficiently the search space and reduce the running time.

5.3.1 Taxonomy of Parallel Tabu Search Algorithm

Several classifications of the parallel TS methods exist in the literature. Similarly to the Branch and Bound method, Trienkens and Bruin [72] classified the parallelization of the TS method in two groups namely *low level parallelization* and *high level parallelization*. As compared to the sequential TS version, the low-level parallelization does not change the way in which the search space is explored, it is only faster. The high-level parallelization has a different behavior from the sequential version since it is based on parallel TS threads exploring simultaneously the search space. Crainic *et al.* [23] introduced a taxonomy of parallel TS algorithms which remains widely used in the literature as of today. This taxonomy is based on three dimensions. The first one "*control cardinality*" defines the parallel TS trajectory which can be controlled by one processor (*1-control i.e.* the search space is explored in the same way as sequential TS) or distributed among several processors (*P-control*). The second dimension "*Control and communication type*" manages the communication, the organization and the synchronization between the parallel processes. It contains four degrees (*Rigid Synchronization, Knowledge Synchronization, Collegial and Knowledge collegial*) that depend on the way in which the processes handle and share the information between them. The third dimension *"Search Differentiation"* focuses on the starting solution and the search strategy of each parallel process. Four ways are identified: *Same initial Point, Same search Strat-*

egy (SPSS); Same initial Point, Different search Strategies (SPDS); Multiple initial Points, Same search Strategy (MPSS); Multiple initial Points, Different search Strategies (MPDS). For more details about the parallel TS classification, the reader may refer to [23].

5.3.2 The Proposed Low-level Parallelization

Our proposed approach can be seen as a low-level parallelization in which the goal is to speed up the execution time of the TS method without trying to improve the solution quality. This approach aims to exploit the multi-core CPU processors available in almost all recent PCs. This parallel approach explores the search space in the same way as a serial TS algorithm. i.e. One-control model is considered since both serial and parallel approaches have the same trajectory in the search space; however, the parallel approach moves faster. Indeed, the serial approach evaluates all neighboring solutions sequentially, one neighbor solution at a time, and then chooses the best one in terms of solution quality to move on. The parallel approach here exploits the fact that each neighbor solution can be evaluated independently from the others. Hence, it uses several CPU threads to evaluate all neighboring solutions simultaneously.

5.3.3 The Proposed High-level Parallelization Schemes

In this section, we describe the proposed parallel TS approaches that exploit the computing power offered by a cluster-based supercomputer. As already mentioned in the parallel TS taxonomy, the parallel TS implementations can be described by the following aspects.

Control Carnality and Communication

The proposed parallelization can be seen as a high-level parallelization in which the search space is explored simultaneously by using several parallel processes, where each process uses the recovery strategy and the neighborhood function defined earlier in this chapter. Therefore, a *P-control* model is considered where each process has its own path which allows exploring efficiently the search space. *e.g.* Using 100 parallel processes, the explored search space is 100 times bigger than the sequential version which may allow to improve the solution quality. In our implementation, we opted for the Rigid synchronization model in which there are no communications between processes. In this model each parallel process manage locally its own tabu list. The goal here is to diversify the search and to prevent the parallel

processes from falling in the same area of the search space; hence, taking the same path. The end of the parallel TS is reached when all the processes have completed their iterations. Before that, each process sends its best-explored solution to the master process (process with id=0) in order to report the final result of the parallel TS method.

Search Differentiation

As seen before, Three possible implementations of the parallel TS algorithm exists and each one of them varies according to the used *initial solution* and the used *search strategy*. The goal of experimenting several implementations is to report the best way to explore the BJSS search space.

In our first implementation, named PTS_SPMS, all parallel processes use the same initial solution and the search trajectory for each process differs according to the used heuristics in the recovery process. *i.e.* Multiple search strategies are used.

In our second implementation, named PTS_MPSS, the search space is divided between all parallel processes, i.e multiple initial points are considered. In addition, all parallel processes use the same search strategy (search trajectory), which is based on the AMCC heuristic in the reconstruction of the neighboring solutions.

Our Third implementation which is named PTS_MPDS uses the same model as PTS_MPSS except that the search strategy differs from one parallel process to another according to the used heuristics in the recovery process.

fInitial solution: In the same way as a sequential TS algorithm, the parallel TS algorithm is able to reach a good solution regardless of the quality of the initial solutions. A very simple way to generate the initial solution, in both BWS and BNS cases, is to schedule the jobs sequentially in a random way. In our parallel approaches, we considered two types of initial solutions. Either all parallel processes begin with the same solution, therefore, a single initial point is considered. Or each parallel process creates its own initial solution generated randomly which allows to split the search space which means that multiple initial points are considered.

Search Strategies: The search strategy aims to define the search trajectory of the parallel TS algorithm. In this work, two search strategies are used according to the used heuristics in the recovery step. In the first strategy, all parallel processes use the same heuristic (AMCC) in the recovery step; therefore, a single strategy is used. We have chosen the AMCC heuristic since it outperforms all other heuristics in terms of obtained solution quality (as shown in

Table 5.4.3). The second strategy combines all the five heuristics in a random manner to allow each parallel process to have its own path in the search space. Therefore, at each iteration of the recovery step, each parallel process chooses a random heuristic to select an unselected alternative pair in order to feasibly complete the neighbor solution. To give better chance to good heuristics to be chosen we assign to them a higher probability. Five heuristics are used in order to feasibly complete the neighboring solutions and define the search trajectory of each parallel processes. In the following, we describe the five heuristics used to feasibly complete s':

- AMCC and SMSP heuristics: these heuristics were defined by Mascis and Pacciarelli [54]. The AMCC heuristic focuses on the solution quality by avoiding the unselected alternative arc that increases the most the *Makespan*. i.e. The AMCC selects the pair $((l,m),(e,f))$ such that $l(o,e) + a_{ef} + l(f,n) = Max\{\ l(o,u) + a_{uv} + l(v,n)\}\ \forall (u,v) \in A$. Therefore $s' := s' \cup (l,m)$. In the same way, the SMSP heuristic selects the pair maximizing the quantity: $l(o,h) + a_{hk} + l(k,n) + l(o,i) + a_{ij} + l(j,n)$ and chooses the arc (i, j) if $l(o,h) + a_{hk} + l(k,n) >= l(o,i) + a_{ij} + l(j,n)$. Therefore $s' := s' \cup (i,j)$.

- Random heuristic: This heuristic consists to choose a random pair $((l,m),(e,f)) \in A$ and selects a random arc (l,m). *i.e.* $s' := s' \cup (l,m)$.

- SPT and LPT heuristics: We have already seen that each alternative pair represents the processing order between two concurrent operations. Let us define C as the set of concurrent operations relative to the unselected alternative pairs in A. The Shortest Processing Time heuristic (SPT) chooses the arc (l',m) relative to the operation o_l which has the shortest processing time in C. The Longest Processing Time (LPT) chooses the arc (l',m) relative to the operation o_l which has the longest processing time in C.

5.4 Experiments

In this section, computational results are given using benchmarks obtained from the classical job shop scheduling instances by dropping the infinite buffer capacity constraint and replacing it with a zero buffer capacity. We tested our algorithms using 53 benchmarks: La01-40 proposed by Lawrence [52] in 1984, Mt10 by Fisher and Thompson [30] (1963), Abz5-6 by Adams, Balas and Zawack [1] (1988), and orb01-10 by Applegate and Cook [5] (1991). The different instances are designated by $n \times m$, where n and m represent respectively the number

of jobs and the number of machines. The approaches have been implemented using C++ language and runs using a machine with 2GHz CPU under Linux operating system.

5.4.1 Sequential TS Results

In the following, we experiment the ability of the proposed TS method in solving both BJSS cases (BWS, BNS). In order to prove the efficiency of our TS method, we first compare our results with the optimal solution of the 18 benchmark instances reported in [54]. After that, we compare our results with the best-literature results known to us for both BJSS cases (BWS, BNS). In the literature, researchers perform up to thirty execution in order to measure the real performance of metaheuristics. Since the deviation of our results from one execution to another one is very low, we perform ten executions of the proposed TS algorithm for each instance. Each execution was limited to 100,000 iterations for the instance sizes: 10 $\times$ 5, 15 $\times$ 5, 20 $\times$ 5, 10 $\times$ 10, and 50,000 iterations for the bigger size instances. The idea here is to have a similar execution time as the literature approach in order to perform a fair comparison.

In order to measure the efficiency of our TS method, named TS_{IN}, we first compare our results with the optimal solutions in [54].

Table 5.4.1 is used to compare our obtained results, for both BJSS cases, with the optimal results of the 10 $\times$ 10 benchmark instances used in [54]. The first column gives the instance name. Column $C_{optimal}$ reports the optimal *Makespan* computed by the B&B algorithm in [54]. Columns C_{mean} and C_{best} report respectively the average makespan and best makespan found by our TS method in ten executions. Finally, the mean relative error(deviation) *MRE* (%) indicates the average deviation of our makespan results from the optimal solutions. For each instance, the bold results indicate that our TS method reaches the optimal solution found by the Branch and Bound method in [54].

The first observation from Table 5.4.1 is the convergence of our TS results to the optimal solutions obtained by the B&B algorithm. Indeed, over the 18 (10 $\times$ 10) benchmark instances, we have been able to reach the optimal solutions for *13* instances in the BWS problem and *16* instances in the BNS case. these results confirm the efficiency of our proposed neighborhood function. Taking into account that each of these instances takes hours to be solved optimally by the B&B algorithm which is not the case of our TS method. The other interesting observation from Table 5.4.1 is the low value of the MRE which indicates the deviation of

Table 5.4.1: Comparison of TS *Makespan* results with the optimal results of the 10 × 10 benchmark instances.

Instance	BWS			BNS		
	$C_{optimal}$	C_{mean}	C_{best}	$C_{optimal}$	C_{mean}	C_{best}
Abz5	1468	1487	**1468**	1641	**1641**	**1641**
Abz6	1145	1190	1160	1249	1254	**1249**
Mt10	1068	1095	1071	1158	1168	**1158**
Orb1	1175	1182	**1175**	1256	1265	1259
Orb2	1041	1062	**1041**	1144	1154	1146
Orb3	1160	1162	**1160**	1311	**1311**	**1311**
Orb4	1146	1187	**1146**	1246	**1246**	**1246**
Orb5	995	1008	**995**	1203	**1203**	**1203**
Orb6	1199	**1199**	**1199**	1266	**1266**	**1266**
Orb7	483	490	**483**	527	535	**527**
Orb8	995	1003	**995**	1139	**1139**	**1139**
Orb9	1039	1076	1045	1130	1131	**1130**
Orb10	1146	**1146**	**1146**	1367	**1367**	**1367**
La16	1060	1090	**1060**	1148	1151	**1148**
La17	929	943	930	968	979	**968**
La18	1025	1038	**1025**	1077	1087	**1077**
La19	1043	1067	1053	1102	1104	**1102**
La20	1060	1085	**1060**	1118	1125	**1118**
MRE		1.7%	0.18%		0.36%	0,02%

our results from the optimal solutions. Indeed, the deviation of our best makespan results (C_{best}) from the optimal solutions varies between 0.02% and 0.18% for both BJSS cases. Moreover, the deviation of our average makespan (C_{mean}) from the optimal solutions is 1.7% for the BWS and 0.36% for the BNS case, which confirms the results quality of our proposal.

Figure 5.4.1 shows the deviation of our TS results over time. The figure shows that the Relative Error (RE) for the La17 instance drops from 23% to 1.6% after 120s, before reaching less than 0.1% after 700 seconds. Two phases can be identified in this figure. A first phase between 5 and 120 seconds characterized by a rapid decrease in the RE. This can be explained

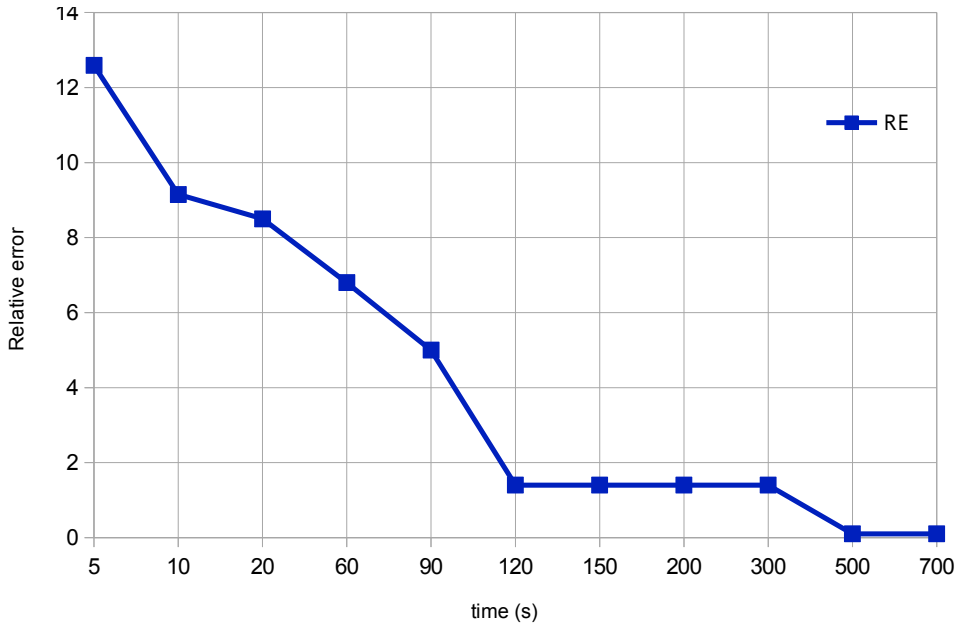


Figure 5.4.1: Variation of the relative errors from the optimal solution over times for La17 instance.

by the ability of our proposed neighborhood to generate good neighboring solutions. After that, begins a second phase in which the RE decreases slowly toward zero, i.e, reaching a near optimal solution.

In the following, we compare the quality of our obtained results with the best-known solutions to our knowledge. For that, we consider Lawrence benchmarks which are the most used in the literature. Table 5.4.2 summarizes our obtained makespan results, the first two columns report the name and size of each instance. For both BJSS cases (BWS and BNS), Column *Best* shows the evaluation and the reference of the best-known solution to our knowledge. Columns C_{mean} and C_{best} report, respectively, the average and best Makespan found by our Tabu search method (TS_{IN}) over ten executions. Finally, Column TS_{gr} represents the evaluation of the best solutions found by the TS method proposed by Gröflin and Klinkert in [42].

For each instance, the bold results indicate that our approach (TS_{IN}) reaches the best-known solution and bold underlined results indicate that our approach improves the best-known solution in the literature. Among the most recent works on the BWS we mention the TS method (TS_{gr}) proposed by Groflin *et al.* [42, 43], *IFS* algorithm and *CP-OPT* method

Table 5.4.2: Our TS makespan results Vs. the best known solutions for both BJSS cases (BWS,BNS).

Instance	*Size*	BWS				BNS		
		Best	TS_{gr}	C_{mean}	C_{best}	*Best*	C_{mean}	C_{best}
La01	10×5	[61, 63] 793	820	780	**793**	[55, 63] 881	**881**	**881**
La02	10×5	[42, 61, 63] 793	793	**793**	**793**	[55, 63] 900	901	**900**
La03	10×5	[61, 63] 715	740	**715**	**715**	[55, 63] 808	810	**808**
La04	10×5	[61, 63] 743	764	**743**	**743**	[55, 63] 859	864	**859**
La05	10×5	[61, 63] 664	666	670	**664**	[55, 63] 732	**732**	**732**
La06	15×5	[61] 1064	1180	1120	1076	[63] 1225	**1207**	**1194**
La07	15×5	[63] 1020	1084	1052	1029	[63] 1133	1138	**1130**
La08	15×5	[61] 1062	1125	1089	**1060**	[55] 1216	**1203**	**1173**
La09	15×5	[63] 1162	1223	1209	1188	[63] 1311	1324	1314
La10	15×5	[61] 1110	1203	1142	**1110**	[63] 1237	1252	**1232**
La11	20×5	[61] 1466	1584	1504	1475	[63] 1641	**1621**	**1577**
La12	20×5	[63] 1271	1391	1326	1276	[63] 1465	**1448**	**1401**
La13	20×5	[61] 1465	1541	1474	**1456**	[63] 1627	**1586**	**1547**
La14	20×5	[61] 1506	1620	1520	**1472**	[63] 1686	**1648**	**1586**
La15	20×5	[61, 63] 1517	1630	1526	**1490**	[63] 1680	**1651**	**1620**
La16	10×10	[63] 1060	1142	1090	**1060**	[55, 63] 1148	1151	**1148**
La17	10×10	[63] 929	977	943	930	[55, 63] 968	979	**968**
La18	10×10	[63] 1025	1078	1038	**1025**	[55, 63] 1077	1087	**1077**
La19	10×10	[63] 1043	1093	1067	1053	[55, 63] 1102	1104	**1102**
La20	10×10	[63] 1060	1154	1085	**1060**	[55, 63] 1118	1125	**1118**
La21	15×10	[63] 1490	1554	1499	**1467**	[63] 1627	**1540**	**1501**
La22	15×10	[63] 1339	1458	1369	1347	[63] 1426	**1421**	**1368**
La23	15×10	[63] 1445	1570	1477	**1442**	[63] 1574	1564	**1537**
La24	15×10	[63] 1434	1546	**1433**	**1398**	[63] 1502	1510	**1447**
La25	15×10	[63] 1392	1499	1420	**1373**	[63] 1533	**1497**	**1453**
La26	20×10	[63] 1989	2125	**1950**	**1929**	[63] 2146	**2023**	**1968**
La27	20×10	[63] 2017	2175	**2007**	**1960**	[63] 2191	**2113**	**2047**
La28	20×10	[61] 2027	2071	**1959**	**1880**	[63] 2245	**2090**	**2046**
La29	20×10	[63] 1846	1990	**1846**	**1803**	[63] 2030	**1942**	**1857**
La30	20×10	[63] 2049	2097	**1982**	**1965**	[63] 2242	**2090**	**2033**
La31	30×10	[61] 2921	3137	**2790**	**2715**	[63] 3219	**2978**	**2942**
La32	30×10	[61] 3237	3316	**3019**	**2987**	[63] 3567	**3163**	**3114**
La33	30×10	[61] 2844	3061	**2755**	**2672**	[63] 3201	**2873**	**2845**
La34	30×10	[61] 2848	3146	**2800**	**2729**	[63] 3202	**2959**	**2862**
La35	30×10	[61] 2923	3171	**2828**	**2776**	[55] 3373	**2961**	**2871**
La36	15×15	[63] 1755	1919	**1741**	**1713**	[63] 1835	**1796**	**1767**
La37	15×15	[63] 1870	2029	**1840**	**1802**	[63] 1931	**1917**	**1871**
La38	15×15	[61] 1708	1828	**1652**	**1630**	[63] 1813	**1770**	**1747**
La39	15×15	[63] 1731	1882	**1719**	**1697**	[63] 1811	**1791**	**1758**
La40	15×15	[63] 1743	1925	**1730**	**1692**	[63] 1815	**1802**	**1780**

proposed by Oddi et al. in [61], and finally, the Iterated Greedy (IG) proposed by Pranzo and Pacciarelli [63]. To the best of our knowledge these results represent the best-known solutions to date.

We first compare our results with the results found by Groflin *et al.* [42, 43] (TS_{gr}). As shown in Table 5.4.2, the results of our proposed approach improve almost all the results of Gröflin et al.. For the La02 instance, both methods converge to the same result (793). Furthermore,

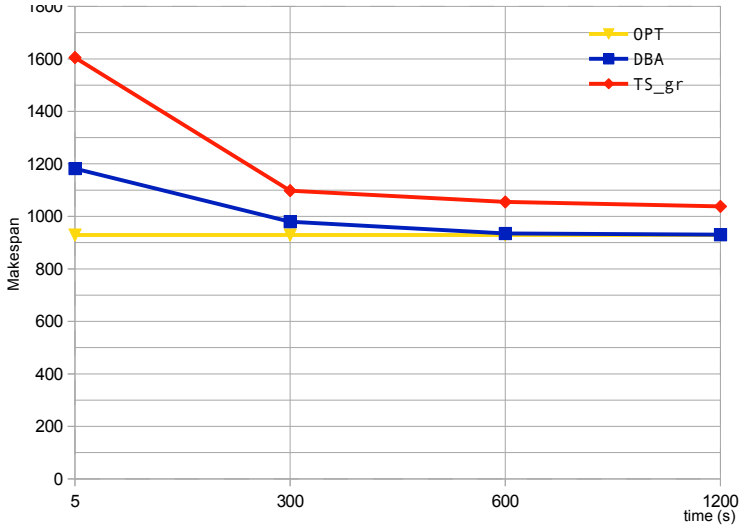


Figure 5.4.2: The behaviour of both TS methods over times for the La17 instance.

Figure 5.4.2 shows the behaviour of both our proposed TS method (blue color) and the TS method proposed by Groflin and Klinkert [42] (Red color) over time. Since we do not have the exact intermediate results of TS_{gr} [42], we use the average gap of the intermediate results for the 10×10 instances to calculate the approximate intermediate results for the La17 instance. The figure shows that the curve of both methods decreases rapidly before stabilizing between 300 and 1200 seconds. We also notice that unlike TS_{gr}, our proposed method converges to the optimal solution (OPT) after 600s only. This is the result of the adaptation of the neighborhood function to the blocking case and the use of the AMCC heuristic that creates good neighboring solutions. The figure shows also the efficiency of our proposed neighborhood function and confirms that the quality of the TS method depends essentially on the quality of the used neighborhood function.

Table 5.4.2 shows a large improvement of the best-known results for most instances in the BWS. More precisely, over 40 experimented benchmark instances, our best TS results (C_{best})

(resp. our mean TS results C_{mean}) improve the best-known results for 23 instances (resp. 16 instances) and match the best-known results for 9 benchmark instances (resp. 3 benchmark instances). The deviation of our C_{best} for the unimproved benchmarks varies between 0.1% and 2% and the mean relative error MRE for the unimproved instances is 0.85%.

Despite the large number of application areas for the BNS problem, this special case has been treated by few authors. Among the most recent works on BNS, we mention: the IG metaheuristic proposed by Pranzo and Pacciarelli [63] and the TS algorithm proposed by Mati and Xie [55] referred to as MX. To the best of our knowledge their results represent the best-known solutions in the literature for the BNS case.

The results found for the BNS version confirm the results found earlier for the BWS and show a large improvement of the solution quality over the best-known results. More precisely, over 40 experimented benchmark instances, our best TS results (C_{best}) (resp. our mean TS results C_{mean}) improve the best-known results for 29 instances (resp. 25 instances) and match the best-known solution for ten benchmark instances (resp. 2 benchmark instances). The deviation of our C_{best} for the only unimproved benchmark (La09) is 0.22%. We notice that unlike our TS method, the performance (result quality) of IG, IFS and CP-OPT methods, for both BJSS cases, decreases for the large problem instances. This behavior is caused by the random factor which plays an important role in these methods.

We have seen that the proposed neighborhood gives interesting results, however, the use of a heuristic in the recovery step is a time consuming especially for large size instances. Therefore, the time complexity of our proposed neighborhood function is higher than the complexity of the classical neighborhood functions. The execution time for our approach varies from minutes for small size instances to reach hours for larger benchmark instances. For this reason, we propose in the following parallel TS approaches.

5.4.2 Parallel TS Results

In the following, we evaluate the ability of our parallel TS approaches to improve the running time of the sequential TS method in addition to the explored search space; hence, the solution quality. To perform a fair comparison with the state of the art results, we use the Lawrence instances [52]. These instances are denoted by $n \times m$, where n and m represent respectively the number of jobs and the number of machines. The different TS parallelization approaches have been implemented using the C++ language and Message Passing Interface (MPI) as a communication tool between parallel processes. The experiments have been carried out

Table 5.4.3: Average execution time and makespan results of the sequential *TS* using different heuristics.

Inst.& *Size*		Best	TS_{AMCC}	TS_{SMSP}	TS_{EST}	TS_{Rand}	TS_{SPT}	TS_{LPT}	Time
La31	30×10	[61] 2921	2867	3034	3015	3314	3831	3791	1012
La32	30×10	[61] 3237	3193	3265	3352	3821	4104	3907	1051
La33	30×10	[61] 2844	2732	2951	2995	3273	3497	3688	1068
La34	30×10	[61] 2848	2853	2976	3042	3408	3581	3723	1226
La35	30×10	[61] 2923	2899	2949	3112	3549	3577	3798	1055

using Ibnbadis cluster which has 32 nodes and 16 CPU-Cores each.

We first begin by showing the results quality and the time consumed by the recovery step using different heuristics. After that, we introduce our low-level parallelization scheme that aims to accelerate the TS execution time by using the multi-core CPU processors of a single node. Finally, we report the results of our proposed high-level parallelization approaches that aim to improve the solutions' quality by exploring the search space in parallel.

Table 5.4.3 reports the average execution time and Makespan results of the sequential TS method using the proposed neighborhood function with six different heuristics. These results are obtained by performing 1,000 iterations of our serial TS algorithm for the BWS case using the hardest Lawrence instances (La31-La35). The first column reports the instance name and size. The second column shows the value and the reference of the best known results in the literature. In the subsequent six columns, we present the makespan result of our proposed TS algorithm using six heuristics in the recovery phase. Finally, Column *time* reports for each instance, the average execution time of the six variants of our sequential TS method.

Two main conclusions can be made from Table 5.4.3. The first conclusion is the efficiency and the positive impact of our proposed neighborhood function on the solution quality, but this efficiency depends essentially on the used heuristic. i.e. The results of using the $AMCC$ heuristic in the recovery step outperform the results of all other heuristics and even the best results in the literature. The second obvious conclusion from table 5.4.3 is the huge execution time needed to perform 1,000 iterations which is a very small number of iterations. To allow the method to converge we may need to explore a much larger search space which induces an unacceptable running time due to the use of heuristics in the recovery step of our algorithm. If say we increase the search space by a factor of 100, then, our TS algorithm will take more

than 24 hours of execution time for each instance. To overcome this drawback, we propose the use of parallel architectures to accelerate the TS method and increase the explored search space, hence, the solution quality.

Table 5.4.4: Speedup of our low-level parallel TS approach exploiting the 16 CPU-cores of a single cluster-node.

Size	$T_{Seq.}$	$T_{parallel}$	Acceleration
10×05	6	5	1
15×05	27	13	2
20×05	75	24	3
10×10	29	13	2
15×10	85	23	3.4
20×10	286	60	5
30×10	1108	174	7
15×15	212	51	4.2

Table 5.4.4 shows the results of our low-level TS parallelization. The first column reports the size of the different benchmarks, where each size contains five instances. Column T_{Seq} reports the average execution time of an optimized sequential TS algorithm. Column $T_{parallel}$ shows the execution time obtained by our low-level parallel TS method exploiting 16 CPU-cores. Finally, Column Acceleration reports the obtained speedup. The reported times in this table represent the average execution time needed by each approach to complete 1,000 iterations.

We can see from the table that our low-level TS parallelization allows to improve the running time while exploring the search space exactly like a serial TS algorithm. Therefore, reaching the same solution quality obtained by the serial version. Indeed, this parallel version is up to 7 times faster for large instances. The speed-up depends on the average number of neighboring solutions that are evaluated simultaneously. This explains the increase in speedup while increasing the size of instances. In other words, the larger the size, the more the average number of neighboring solutions and the higher the speedup. This also explains why the speedup is not linear.

In the following, we report the ability of our high level parallel TS approaches to extend the explored search space and to improve the solution quality. To this aim, we fixed the

Table 5.4.5: Comparing our parallel TS makespan results with the sequential results and the state of the art solutions for the BWS problem.

Inst.	*Best*	*Sequential TS*	Parallel TS			RC_{seq}	RC_{Best}
			MPSS	*MPDS*	*SPDS*		
La01	793 [61, 63]	**793**	**793**	**793**	**793**	0.0	0.0
La02	793 [61, 63]	**793**	**793**	**793**	**793**	0.0	0.0
La03	715 [61, 63]	721	**715**	**715**	**715**	-0.8	0.0
La04	743 [61, 63]	**743**	**743**	**743**	**743**	0.0	0.0
La05	664 [61, 63]	**664**	**664**	**664**	**664**	0.0	0.0
La06	1064 [63]	1142	1092	1081	1097	-4.3	3.0
La07	1020 [61]	1082	1026	1046	1072	-5.2	1.0
La08	1062 [63]	1095	**1060**	1076	1081	-3.2	-0.2
La09	1162 [63]	1216	1183	1199	1177	-2.7	3.7
La10	1110 [63]	1179	**1110**	1138	1146	-5.8	0.0
La11	1466 [63]	1525	**1445**	1502	1585	-3.6	-0.3
La12	1271 [61]	1313	**1269**	1315	1327	-3.3	-0.2
La13	1465 [63]	1485	**1440**	1462	1497	-3.0	-2.1
La14	1506 [63]	1540	**1465**	1498	1512	-4.8	-0.1
La15	1517 [61, 63]	1540	**1515**	1543	1558	-1.6	-0.2
La16	1060 [63]	1122	**1060**	1103	1076	-5.5	0.0
La17	929 [63]	942	930	940	940	-1.2	0.1
La18	1025 [63]	1056	**1025**	1038	1055	-2.9	0.0
La19	1043 [63]	1068	**1043**	1073	1068	-2.3	0.0
La20	1060 [63]	1092	1074	1110	1095	-1.6	1.3
La21	1490 [61]	1506	**1410**	1566	1555	-6.3	-1.6
La22	1339 [63]	1397	**1303**	1380	1399	-6.7	-2.7
La23	1445 [63]	1501	**1338**	1533	1515	-10.8	-6.8
La24	1434 [61]	**1430**	**1399**	1471	1478	-2.1	-2.0
La25	1392 [61]	1453	**1343**	1356	1476	-7.5	-3.5
La26	1989 [63]	2014	**1914**	2002	1998	-4.9	-3.8
La27	2017 [61]	2034	**1959**	2067	2052	-3.6	-2.3
La28	2027 [63]	**1997**	**1878**	2022	1982	-5.9	-7.0
La29	1846 [61]	1849	**1819**	1913	1891	-1.6	-1.4
La30	2049 [63]	**1994**	**1942**	2058	2013	-2.6	-5.2
La31	2921 [61]	**2813**	**2748**	2915	2880	-2.3	5.9
La32	3237 [61]	**3064**	**2935**	3136	3118	-4.2	-9.3
La33	2844 [61]	**2709**	**2662**	1819	2849	-1.7	-6.4
La34	2848 [61]	**2823**	**2731**	2913	2877	-3.2	-4.1
La35	2923 [61]	**2850**	**2722**	2950	2910	-4.5	-6.9
La36	1755 [61]	1787	**1685**	1822	1872	-5.7	-0.9
La37	1870 [61]	**1838**	**1815**	1943	1927	-1.2	-1.8
La38	1708 [63]	**1683**	**1606**	1765	1723	-4.8	-6.0
La39	1731 [61]	1748	**1705**	1843	1841	-2.4	-0.5
La40	1743 [61]	1729	**1708**	1833	1820	-1.2	-0.4

Table 5.4.6: Comparing our obtained makespan results with the best-known solutions for the BNS problem.

Inst.	*Best*	*sequential TS*	Parallel TS			RC_{seq}	RC_{Best}
			MPSS	*MPDS*	*SPDS*		
La01	881[55, 63]	**881**	**881**	**881**	**881**	0,0	0,0
La02	900[55, 63]	**900**	**900**	**900**	**900**	0,0	0,0
La03	808[55, 63]	**808**	**808**	**808**	**808**	0,0	0,0
La04	859[55, 63]	**859**	**859**	**859**	**859**	0,0	0,0
La05	732[55, 63]	**732**	**732**	**732**	**732**	0,0	0,0
La06	1225[63]	1254	**1199**	**1207**	**1199**	-4,4	-2,1
La07	1133[63]	**1133**	**1130**	**1133**	1138	-0,3	-0,3
La08	1216[55]	**1191**	**1173**	**1173**	**1196**	-1,5	-3,5
La09	1311[63]	1352	**1311**	1313	1314	-3,0	0,0
La10	1237[63]	**1237**	**1222**	1241	1252	-1,2	-1,2
La11	1641[63]	**1625**	**1591**	**1635**	**1629**	-2,1	-3,0
La12	1465[63]	1471	**1406**	**1438**	**1399**	-4,4	-4,0
La13	1627[63]	**1615**	**1553**	**1585**	**1538**	-3,8	-4,5
La14	1686[63]	**1671**	**1608**	**1650**	**1544**	-3,8	-4,6
La15	1680[63]	**1666**	**1616**	**1652**	**1640**	-3,0	-3,8
La16	1148[55, 63]	**1148**	**1148**	1158	1169	0,0	0,0
La17	968[55, 63]	999	**968**	995	**968**	-3,1	0,0
La18	1077[55, 63]	1080	**1077**	**1077**	**1077**	-0,3	0,0
La19	1102[55, 63]	**1102**	**1102**	1113	**1102**	0,0	0,0
La20	1118[55, 63]	1156	**1118**	**1118**	1136	-3,3	0,0
La21	1627[63]	**1595**	**1504**	**1558**	**1574**	-5,7	-7,6
La22	1426[63]	1466	**1384**	**1420**	1447	-5,6	-2,9
La23	1574[63]	1622	**1534**	**1560**	**1566**	-5,4	-2,5
La24	1502[63]	1559	**1485**	1546	1563	-4,7	-1,1
La25	1533[63]	1539	**1476**	1544	**1505**	-4,1	-3,7
La26	2146[63]	**2082**	**2012**	**2075**	**2105**	-3,4	-6,2
La27	2191[63]	**2076**	**2074**	**2159**	**2185**	-0,1	-5,3
La28	2245[63]	**2072**	**2051**	**2109**	**2081**	-1,0	-8,6
La29	2030[63]	**1915**	**1914**	**1972**	**2026**	-0,1	-5,7
La30	2242[63]	**2135**	**2034**	**2152**	**2137**	-4,7	-9,3
La31	3219[63]	**3009**	**2866**	**3003**	**3029**	-4,8	-11,0
La32	3567[63]	**3234**	**3040**	**3288**	**3276**	-6,0	-14,8
La33	3201[63]	**2871**	**2803**	**2959**	**2951**	-2,4	-12,4
La34	3202[63]	**2946**	**2876**	**3008**	**3038**	-2,4	-10,2
La35	3373[55]	**2958**	**2925**	**3067**	**3086**	-1,1	-13,3
La36	1835[63]	**1823**	**1757**	1912	1859	-3,6	-4,3
La37	1931[63]	1959	**1880**	2046	2064	-4,0	-2,6
La38	1813[63]	**1809**	**1716**	1844	1816	-5,1	-5,4
La39	1811[63]	1837	**1750**	1926	1908	-4,7	-3,4
La40	1815[63]	**1773**	**1742**	1888	1894	-1,7	-4,0

number of iterations for both the sequential and the parallel approaches to 10,000 iterations for [La01-La20] instances and 5,000 iterations for [La21, La40] instances. We first begin by comparing the quality of our obtained parallel results with the results of the serial TS version in addition to the best results in the literature.

Tables 5.4.5 and 5.4.6 show the makespan results of our parallel TS approaches for both BWS and BNS problems. Column *Sequential TS* reports the best makespan obtained by our serial TS method over four executions. Columns MPSS, MPDS, and SPDS show the results of our proposed TS parallelization approaches. Each approach exploits 240 CPU-cores (240 parallel processes are used) and uses the same iteration number as the serial version. Column RC_{seq} shows the relative change of our best parallel results from our serial TS version using the following formula: $\mathtt{RC}_{\mathtt{seq}} = \frac{\mathtt{MPSS-Seq}}{\mathtt{Seq}} \times 100$. Similarly, Column RC_{best} shows the relative change of our best parallel results from the best results in the literature using the following formula: $\mathtt{RC}_{\mathtt{best}} = \frac{\mathtt{MPSS-best}}{\mathtt{best}} \times 100$. The negative values obtained by these formulas indicate an improvement against the sequential results (resp. The best results in the literature.) For each instance, the bold results indicate that our approaches reach the best-known solution and bold underlined results indicate that our approach improves the current best solution which is a very difficult goal to achieve.

First of all, both tables 5.4.5, and 5.4.6 shows the good results of our sequential version which is based on the AMCC heuristic in the recovery step. Even with a such low number of iterations, the serial version allows us to improve the best results for **10** instances for the BWS (resp. **19** instances for BNS problem) and reach the current best results for **4** instances for the BWS (resp.**9** for BNS problem). Increasing the number of iterations of the sequential version generally improves the quality but the time needed for each instance increases unacceptably especially for large size instances. We have two options, either waiting a huge amount of time for each instance to finish or taking advantage of the parallel TS method, which can simply explore several areas of the search space simultaneously. Therefore exploring a huge search space within the same execution time as the serial TS version.

The results of our first parallel approach (MPSS) based on multiple initial solutions and single search strategy using only the AMCC heuristic in the recovery step outperform the results of the MPDS and SPDS approaches which are based on different search strategies using a single or multiple initial solutions. The used strategies in the MPDS and SPDS approaches are obtained by randomly choosing at each step of the reconstruction phase one heuristic to select which arc to pick first until a complete neighbor solution is obtained.

We can reach two main conclusions from table 5.4.5 and table 5.4.6. The first conclusion is that the results of the parallel Tabu search method depend essentially on the quality of the neighboring solutions. Indeed, the results quality of our parallel TS approaches depend essentially on the used heuristics in the recovery step, since these heuristics guide the search trajectory of our parallel TS method. The neighbour solutions produced by the AMCC heuristic are better than the neighbour solutions produced by combining the five heuristics randomly. For this reason, the results of the MPSS approach are much better than the MPDS and the SPDS approaches. The second conclusion is the positive impact of the parallelization on the solution quality. Using the same running time as the serial version, our parallel approach (MPSS) improves almost all the results of the serial version for both BJSS cases as shown in the Column RC_{seq} of tables 5.4.5 and 5.4.6.

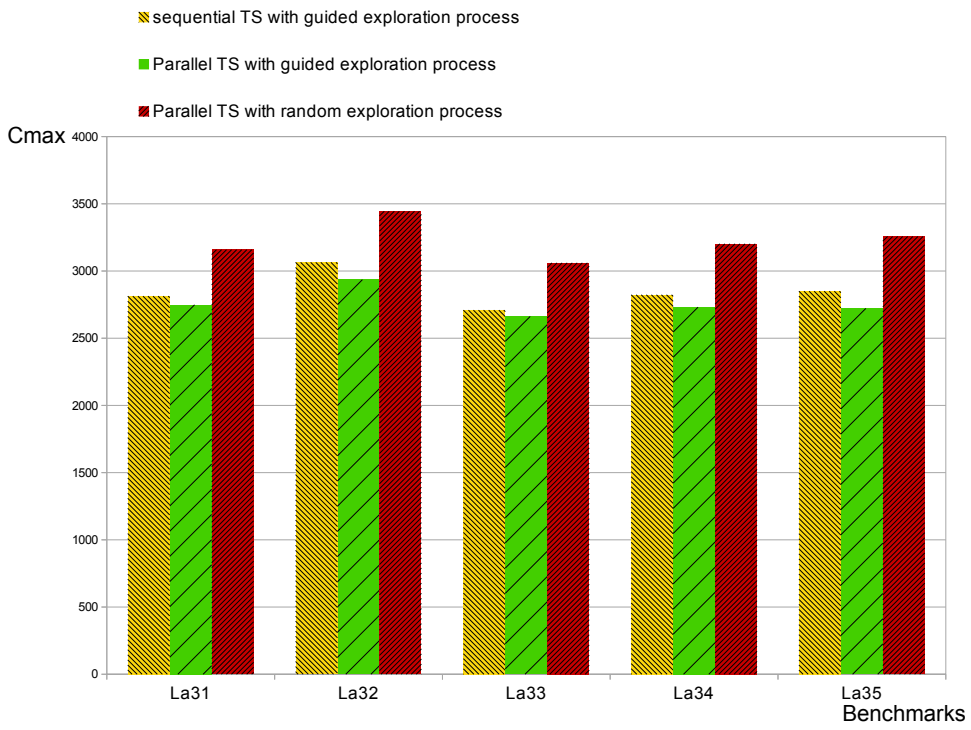


Figure 5.4.3: Comparing the makespan results between the sequential TS and two parallel TS approaches.

In-addition, Column RC_{best} of Table 5.4.5 and Table 5.4.6 shows a significant improvement of the best-known results, especially for large instances. More precisely, over 40 benchmark instances, our parallel approach for the BWS problem (resp. BNS problem) improves on the

best-known results for **26** instances (resp. **29** instances) and matches the best known results for **9** instances (resp. **11** instances). This improvement proves the ability of our parallel TS algorithm not only to find good solutions, but also in a short amount of time. The deviation of our results from the best results in the literature, for the unimproved instances, varies between 0% and 3% for the BWS problem. For the BNS problem, our results improve the results of almost all instances.

Table 5.4.7: Makespan results of the parallel and sequential TS methods for both BJSS cases (BWS, BNS).

Sequential TS: 50,000 iterations. (best of 10 executions)
Parallel TS (PTS): 5,000 iterations.(only one execution)

Instance	*Size*	BWS		BNS	
		PTS	Seq_{TS}	*PTS*	Seq_{TS}
La21	15×10	**1410**	1467	1504	1501
La22	15×10	**1303**	1347	1384	1368
La23	15×10	**1338**	1442	**1534**	1537
La24	15×10	**1398**	1398	1485	1447
La25	15×10	**1343**	1373	1476	1453
La26	20×10	**1914**	1929	2012	1968
La27	20×10	**1959**	1960	2074	2047
La28	20×10	**1878**	1880	2051	2046
La29	20×10	1819	1803	1914	1857
La30	20×10	**1942**	1965	2034	2033
La31	30×10	2748	2715	**2886**	2942
La32	30×10	**2935**	2987	**3040**	3114
La33	30×10	**2662**	2672	**2803**	2845
La34	30×10	2731	2729	2876	2862
La35	30×10	**2722**	2776	2925	2871
La36	15×15	**1685**	1713	**1757**	1767
La37	15×15	1815	1802	1880	1871
La38	15×15	**1606**	1630	**1716**	1747
La39	15×15	1705	1697	**1750**	1758
La40	15×15	1708	1692	**1742**	1780

Figure 5.4.3 compares the makespan results between two parallel approaches for the hardest Lawrence instances (La31-La35). The first approach uses a guided search process based on

the AMCC heuristic to rebuild the neighbour solutions while the second approach uses a random search based on the random heuristic for the same purpose. The horizontal line shows the used instances and the vertical line reports the obtained makespan (the smaller is the better).

The goal of the parallelization is to increase the explored search space in order to improve the solution quality. This goal can not be achieved when a random exploration strategy is used to explore the search space. The figure shows that the result quality of the parallel TS with a random exploration strategy is even worse than a guided serial TS version. The figure also shows that combining parallelization with a guided exploration strategy allows to improve the solution quality. In other words, it is not interesting to use parallelization to improve the solution quality when a random search is performed.

Table 5.4.7 shows a comparison between the sequential results (*Seq. TS*) with 50,000 iterations (best of ten executions) and the parallel result (*PTS*) with only 5,000 iterations (only one execution is performed.) for both BJSS cases. The time needed by our parallel TS for each instance is **100 times less** than the time needed by sequential TS. Even with such small running time as compared with the sequential approach, our parallel results with only one execution are in most cases better than the serial results. This is understandable since the parallel approach explores several areas simultaneously, therefore, more chance to find good solutions. For this reason, it is more beneficial to use parallelization than increasing the number of iterations for the sequential version.

5.4.3 Comparing the Approximate Results of our Parallel B&B Algorithm with the Best Results of our Parallel TS Algorithm

In this section, we aim to evaluate the strengths and weaknesses of each proposed approach. Fore that, we compare the approximate results of our parallel B&B algorithm with the results of our parallel TS algorithm using the Lawrence instances. Table 5.4.8 and Table 5.4.9 summarize the obtained results for both BJSS cases, i.e., BJSS with swap allowed (BWS), and BJSS with no swap allowed (BNS). For both tables, the first two columns report the instance name and size. Column three (*Reference*) shows the reference to the best-known solutions reported in column four (Best). Column $B\&B_{Parallel}$ reports the approximate results found by our parallel B&B algorithm using 320 CPU-cores for 7200 seconds (one execution

Table 5.4.8: Parallel $B\&B$ approximate results VS. Parallel TS results for the BWS problem.

Instance	*Size*	*Reference*	*Best*	$TS_{parallel}$	$B\&B_{parallel}$
La01	10 × 5	IFS [61], IG [63]	793	**793**	**793**
La02	10 × 5	IFS [61], IG [63]	793	**793**	**793**
La03	10 × 5	IFS [61], IG [63]	715	**715**	**715**
La04	10 × 5	IFS [61], IG [63]	743	**743**	**743**
La05	10 × 5	IFS [61], IG [63]	664	**664**	**664**
La06	15 × 5	IFS [61]	1064	1092	1060
La07	15 × 5	IG [63]	1020	1026	1016
La08	15 × 5	IFS [61]	1062	1060	1040
La09	15 × 5	IG [63]	1162	1183	1141
La10	15 × 5	IFS [61]	1110	1110	1096
La11	20 × 5	IFS [61]	1466	1445	1465
La12	20 × 5	IG [63]	1271	1269	1253
La13	20 × 5	IFS [61]	1465	1440	1383
La14	20 × 5	IFS [61]	1506	1465	1467
La15	20 × 5	IFS [61], IG [63]	1517	1515	1511
La16	10 × 10	IG [63]	1060	**1060**	**1060**
La17	10 × 10	IG [63]	929	930	929
La18	10 × 10	IG [63]	1025	**1025**	**1025**
La19	10 × 10	IG [63]	1043	**1043**	**1043**
La20	10 × 10	IG [63]	1060	1074	1060
La21	15 × 10	IG [63]	1490	1410	1433
La22	15 × 10	IG [63]	1339	1303	1424
La23	15 × 10	IG [63]	1445	1338	1436
La24	15 × 10	IG [63]	1434	1399	1354
La25	15 × 10	IG [63]	1392	1343	1313
La26	20 × 10	IG [63]	1989	1914	1994
La27	20 × 10	IG [63]	2017	1959	2005
La28	20 × 10	IFS [61]	2027	1878	2020
La29	20 × 10	IG [63]	1846	1819	1725
La30	20 × 10	IG [63]	2049	1942	2055
La31	30 × 10	CP_OPT [61]	2921	2748	3028
La32	30 × 10	CP_OPT [61]	3237	2935	3368
La33	30 × 10	CP_OPT [61]	2844	2662	3037
La34	30 × 10	CP_OPT [61]	2848	2731	2961
La35	30 × 10	CP_OPT [61]	2923	2722	3065
La36	15 × 15	IG [63]	11755	1685	1701
La37	15 × 15	IG [63]	1870	1815	1848
La38	15 × 15	IFS [61]	1708	1606	1598
La39	15 × 15	IG [63]	1731	1705	1714
La40	15 × 15	IG [63]	1747	1708	1714

Table 5.4.9: Parallel *B&B* results vs. TS parallel results for the BNS problem.

Instance	*Size*	*Reference*	*Best*	$TS_{Parallel}$	$B\&B_{Parallel}$
La01	10 × 5	MX [55], IG [63]	881	**881**	**881**
La02	10 × 5	MX [55], IG [63]	900	**900**	**900**
La03	10 × 5	MX [55], IG [63]	808	**808**	**808**
La04	10 × 5	MX [55], IG [63]	859	**859**	**859**
La05	10 × 5	MX [55], IG [63]	732	**732**	**732**
La06	15 × 5	IG [63]	1225	1199	1194
La07	15 × 5	IG [63]	1133	1130	1127
La08	15 × 5	MX [55]	1216	**1173**	**1173**
La09	15 × 5	IG [63]	1311	1311	1309
La10	15 × 5	IG [63]	1237	1222	1218
La11	20 × 5	IG [63]	1641	1591	1587
La12	20 × 5	IG [63]	1465	1406	1410
La13	20 × 5	IG [63]	1627	1553	1569
La14	20 × 5	IG [63]	1686	**1608**	**1608**
La15	20 × 5	IG [63]	1680	1616	1639
La16	10 × 10	IG [63], MX [55]	1148	**1148**	**1148**
La17	10 × 10	IG [63], MX [55]	968	**968**	**968**
La18	10 × 10	IG [63], MX [55]	1077	**1077**	**1077**
La19	10 × 10	IG [63], MX [55]	1102	**1102**	**1102**
La20	10 × 10	IG [63], MX [55]	1118	**1118**	**1118**
La21	15 × 10	IG [63]	1627	1504	1539
La22	15 × 10	IG [63]	1426	1384	1391
La23	15 × 10	IG [63]	1574	1534	1548
La24	15 × 10	IG [63]	1502	1485	1469
La25	15 × 10	IG [63]	1533	1476	1451
La26	20 × 10	IG [63]	2146	2012	2118
La27	20 × 10	IG [63]	2191	2074	2179
La28	20 × 10	IG [63]	2245	2051	2203
La29	20 × 10	IG [63]	2030	1914	2026
La30	20 × 10	IG [63]	2242	2034	2185
La31	30 × 10	IG [63]	3219	2866	3284
La32	30 × 10	IG [63]	3567	3040	3660
La33	30 × 10	IG [63]	3201	2803	3199
La34	30 × 10	IG [63]	3202	2876	3295
La35	30 × 10	MX [55]	3373	2925	3424
La36	15 × 15	IG [63]	1835	1753	1814
La37	15 × 15	IG [63]	1931	1880	1961
La38	15 × 15	IG [63]	1813	1716	1752
La39	15 × 15	IG [63]	1811	1750	1776
La40	15 × 15	IG [63]	1815	1742	1806

only). Finally, Column $TS_{Parallel}$ shows the results of our parallel TS algorithm which is based on the MPSS model, exploiting 240 CPU-cores (One execution only).

Among the best works, in terms of solution quality, for the BJSS with swap we find the IFS and CP-OPT methods proposed by Oddi et al. in [61], and the Iterated Greedy algorithm (IG) proposed by Pranzo and Pacciarelli in [63]. For the BNS case, we find the work of Mati and Xie in [55] referred to as *MX* ,and IG algorithm proposed by Pranzo and Pacciarelli in [63]. For each instance, the bold result indicates that both parallel approaches reach the same makespan value, and the underline result indicates the best makespan value found for this instance.

Table 5.4.8 and Table 5.4.9 show the efficiency of our proposed approximate parallel algorithms in dealing with the high complexity of the BJSS problem. Indeed, the obtained results fully demonstrate the superiority of our proposed parallel approaches on all considered benchmark instances. i.e., For both BJSS cases and over 40 experimented benchmark instances, our approaches reach the best solutions for ten instances, and improve on the rest of them. This performance is explained by (1) the adaptation of these algorithms to the blocking case and (2) the good impact of the parallelization on the solution quality.

Since the results of both Table 5.4.8 and Table 5.4.9 do not show a dominance relation, we can not affirm that one parallel algorithm is better than the other one. For the BWS problem (resp. the BNS problem), we can notice the following. For the small and medium size instances (La01-La20), we notice that the results of the parallel B&B algorithm outperform the results of the parallel TS algorithm. Indeed, the parallel B&B approach is better in ten instances (resp. five instances in the BNS case), and the parallel TS algorithm is better in only two instances (resp. three instances in the BNS case). For the large size instances (La21-La40), the performance of the parallel TS algorithm is much better than the parallel B&B algorithm. Indeed, over 20 instances between La21 and La40, the parallel TS is better in 17 instances (resp. 18 instances in the BNS case).

From these results, we can see that the parallel B&B algorithm is more efficient for small and medium size instances. However, when increasing the size of instances the performance of the latter decreases. This can be explain by the huge size of the B&B search tree inducing a huge difficulty to reach leaf-nodes. i.e. As the search tree get deeper, the ratio of reached leaf-nodes to the explored nodes get smaller; thus the performance of this approach drops significantly. As compared to the parallel B&B approach, the performance of the parallel TS algorithm remains stable when increasing the size of instances.

5.5 Conclusions

In order to deal with large benchmark instances, we proposed in this chapter the adaptation and the parallelization of the Tabu search metaheuristic. The goals are: (1) to solve approximately the BJSS problem and (2) to study the behaviour of parallel metaheuristics. The difficulty of solving the BJSS problem by means of approximate algorithms lies in the huge number of created infeasible solutions during the search. i.e. Due to the impact of the blocking constraint, the classical reproduction techniques (mutation, crossover, and neighborhood function, etc) generate in most cases infeasible solutions. Therefore, a low ratio of explored feasible solution; thus, a low performance in terms of obtained solutions' quality. Indeed, the classical TS neighborhood produces in most cases infeasible solutions. For this reason, we proposed in this chapter the adaptation of the classical TS neighborhood function to the blocking constraint. Our goal is (1) to restore the feasibility of neighboring solutions and (2) guide the search process. The proposed neighborhood consists to apply the classical neighborhood function then recover the feasibility of neighboring solutions by removing some of the arcs that cause the infeasibility and rebuild the neighboring solutions by using heuristics. Unlike other metaheuristics, the TS method shows a stability of performance for the large problem instances which confirms the efficiency of the TS method. However, this efficiency depends strongly on the quality of the used neighborhood function. Our obtained results improve the most of the state of the art results which indicates the efficiency of our proposed neighborhood function. However, using heuristics in the recovery step makes the TS algorithm very slow inducing an unacceptable running time to explore efficiently the search space. To overcome this problem and to report the impact of using parallel architectures on the solutions' quality, we also proposed in this chapter several TS parallelization approaches. The proposed parallelization approaches exploit the computing power offered by a cluster-based supercomputer. The obtained results confirm the efficiency of the proposed parallel approaches that allowed us to improve on most of the state of the art results. We have noticed that the parallelization allows to increase the explored search space leading to an improvement in the solutions' quality. However, the parallelization doesn't allow to improve the solution quality if a random strategy is used to explore the search space. Moreover, our proposed approaches improve the best-known results (new upper bounds) for 23 benchmark instances in the BWS problem and 29 benchmark instances for the BNS case.

6

Conclusions

Heterogeneous parallel architectures (GPUs and Multi-core CPUs) form the basis of all recent computers. They can be found on our personal computers, workstations, clusters, and clouds. Their availability continues to grow since they become more and more efficient and affordable. This fact has attracted researchers to take benefit from the opportunities offered by the available parallelism for the resolution of various problems. The challenge here is to design and implement efficient parallel algorithms that exploit and combine all levels of parallelism. Indeed, the general objective of our thesis is to study the relationship between parallel computing and resolution methods; in order to better understand the benefits that parallelism can bring for solving the Blocking Job shop Scheduling (BJSS) Problem. This problem is encountered in multiple areas including industrial manufacturing with limited storage space, resources scheduling, trains and air traffic scheduling, etc. Due to the importance of the blocking constraint and its huge economic impact, the BJSS problem is relatively well studied in the literature. Indeed, it has been shown that this problem is NP-hard in the strong sense in which only a few special cases of it can be solved in a polynomial time.

The first part of this thesis was dedicated to the implementation and the parallelization

of the Branch and Bound (B&B) algorithm using Multi and Many-core architectures. The goal was to report the impact of the parallelism on solving optimally the BJSS problem. For this purpose, five parallel approaches in addition to the serial version have been proposed. In our serial version, we focused our attention on tuning the selection and exploration strategies since they are the main factors that influence the performance of the B&B algorithm. To accelerate this latter, two low-level parallel approaches exploiting Graphic Processing Units (GPUs) have been proposed. These two approaches aim to accelerate the B&B algorithm by parallelizing the computation of the lower bound which represents more than 85% of the total execution time. The results confirm the high efficiency of our proposed approaches and show a relative speedup of 66x, i.e. Our approaches are up to 66 times faster as compared to our optimized serial B&B algorithm. With the goal of exploiting the Multi-core CPUs, a high-level parallel approach has been proposed. In this approach, the search tree is built in parallel using several threads where each one has its own B&B instance exploiting one CPU-core. With 80% efficiency and using 160 CPU-cores, this approach allowed-us to solve optimally ten benchmark instances that have never been solved before.

In order to take benefit from the power of both the CPU-cores and the GPU at the same time, two hybrid approaches dedicated to heterogeneous platforms have been proposed. These two approaches represent a combination of the high-level CPU approach and the two low-level GPU approaches using Nvidia Multi Processes Service (MPS). This tool allows several host processes to use the GPU simultaneously. The results showed that these approaches are up to 160 times faster compared to our optimized serial version. The good performance of the hybrid approaches is explained by the higher occupation of the GPU over time. i.e. At each moment, these approaches can have several parallel threads executing instructions on the GPU, others send and receive data from the GPU, and yet others perform branching operations on the CPU.

In real-life problems, instances are generally large and could take hundreds of years to be solved optimally, even on massively parallel machines. To deal with such instances, approximate algorithms are needed. These algorithms aim to return a near optimal solution in a reasonable time. The second part of our thesis focused on solving the BJSS problem by using parallel approximate algorithms. Due to the impact of the blocking constraint, approximate algorithms in their classical form are not efficient because of the high ratio of infeasible to explored solutions. The challenge is then to propose original algorithms adapted to the blocking constraint.

Our first proposition was to adapt the parallel B&B algorithm to act as an approximate method by tuning the exploration and selection strategy of parallel processes which allows to explore a huge number of solutions in a small amount of time. The obtained results showed a good improvement over the best results in the literature for more than 52 benchmark instances in both BJSS cases. These experiments demonstrate the high benefit of using the parallel B&B algorithm as an approximate method for the problems with a high ratio of unfeasible to explored solutions.

Our second proposition was an adaptation of the Tabu Search method for the BJSS problem. This metaheuristic has proven its performance for a large number of optimization problems. However, applying it to the BJSS problem produces infeasible solutions in most of the cases. To overcome this drawback, we proposed in this thesis a TS algorithm with a new neighborhood function adapted to the blocking constraint. The proposed neighborhood, based on reconstruction strategy, allowed us to recover the feasibility of neighboring solutions; hence, exploring only feasible solutions. The experiments showed that our obtained results improve over most of the state of the art results which indicate the quality of our proposed neighborhood function.

To study the impact of the parallelization on the behavior of metaheuristics (both the running time and the solution quality), several parallel TS approaches have been proposed. The first parallel TS approach is a low-level parallelization in which the goal is to speedup the running time by evaluating all neighboring solutions simultaneously using several CPU threads. The results showed a good reduction in running time without improving the solution quality since this version explores the search space exactly as a serial TS version. We noticed that performance of this parallel approach in terms of running time is variable since it depends on the number of neighboring solutions at each iteration of the TS algorithm. i.e. The larger the number of neighboring solutions, the better is the performance.

With the goal of improving the solution quality, high-level parallel TS approaches have been proposed. These parallel approaches exploit the computing power offered by a cluster-based supercomputer. These approaches have hundreds of parallel processes exploring the search space using the same or different search strategies according to the used heuristics in the recovery step. The experiments indicate the positive impact of the parallelization on the solution quality. Indeed, the obtained results improve on the most of the state of the art results using only one execution.

Several conclusions can be drawn from our experience with the parallel resolution of the

BJSS problem:

- The main conclusion is the positive impact of the parallelization on both the execution time and the quality of the obtained solution.
- The necessity of combining both high-level and low-level parallel schemes in order to benefit from the computing power of heterogeneous architectures.
- The benefit of considering the parallel B&B algorithm as an approximate method for complex problem where the ratio of infeasibility is very high.
- The obtained results are strongly influenced by the used parallelization model and the goals of the parallelization.
- For metaheuristics, the parallelism allows to increase significantly the explored search space which may allow to improve the solution quality only if a guided search strategy is used. In other words, the parallelism can't help to improve the solution quality if a random exploration strategy is used by the parallel processes.
- It is more benefit in terms of solution quality and running time to use the high-level parallel approach for a small amount of time (minutes) than using the serial approach for a long time (hours). This is logical since the parallel approach explores several paths simultaneously in the search space; therefore, more chance to encounter good solutions.
- The necessity of adapting the metaheuristic reproduction mechanisms to the problem that needs to be solved.

As a perspective to this work, we plan to: (1) propose and implement new hybrid parallel schemes that exploit more heterogeneous computing architectures such as Intel Xeon Phi and Nvidia GPUs. (2) Propose new heuristics and metaheuristics for the BJSS problem in order to deal efficiently with the blocking constraint. (3) Proposing a new lower-bound for the B&B algorithm adapted to the BJSS problem.

References

[1] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.

[2] Abdelhakim AitZai, Brahim Benmedjdoub, and Mourad Boudhar. A branch and bound and parallel genetic algorithm for the job shop scheduling problem with blocking. *International Journal of Operational Research*, 14(3):343–365, 2012.

[3] Sheldon B Akers Jr and Joyce Friedman. A non-numerical approach to production scheduling problems. *Journal of the Operations Research Society of America*, 3(4):429–442, 1955.

[4] Selim G Akl. The design of efficient parallel algorithms. In *Handbook on Parallel and Distributed Processing*, pages 13–91. Springer, 2000.

[5] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2):149–156, 1991.

[6] G Authié, JM Garcia, A Ferreira, JL Roch, G Villard, J Roman, C Roucairol, and B Virot. Parallélisme et applications irrégulières. *Hermès*, 1995.

[7] Philippe Baptiste. *Une étude théorique et expérimentale de la propagation des contraintes de ressources*. PhD thesis, Compiègne, 1998.

[8] Ahcène Bendjoudi, Mehdi Chekini, Makhlouf Gharbi, Malika Mehdi, Karima Benatchba, Fatima Sitayeb-Benbouzid, and Nouredine Melab. Parallel b&b algorithm for hybrid multi-core/gpu architectures. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 914–921. IEEE, 2013.

[9] Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete applied mathematics*, 5(1):11–24, 1983.

[10] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[11] Patrice Roger Calégari. Parallelization of population-based evolutionary algorithms for combinatorial optimization problems. 1999.

[12] Jacques Carlier and Philippe Chrétienne. *Problèmes d'ordonnancement: modélisation, complexité, algorithmes*. Masson, 1988.

[13] Jacques Carlier and Eric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161, 1994.

[14] Tiago Carneiro, Albert Einstein Muritiba, Marcos Negreiros, and Gustavo Augusto Lima de Campos. A new parallel schema for branch-and-bound algorithms using gpgpu. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 41–47. IEEE, 2011.

[15] Imen Chakroun, Nordine Melab, Mohand Mezmaz, and Daniel Tuyttens. Combining multi-core and gpu computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing*, 73(12):1563–1577, 2013.

[16] Imen Chakroun, Mohand Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[17] Bruno Codenotti and Mauro Leoncini. *Introduction to parallel processing (international computer science series)*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[18] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[19] Francesco Corman, Andrea D'Ariano, Dario Pacciarelli, and Marco Pranzo. A tabu search algorithm for rerouting trains during rail operations. *Transportation Research Part B: Methodological*, 44(1):175–192, 2010.

[20] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. Introduction à l'algorithmique. 1994.

[21] NVIDIA Corporation. Multi-process service, 2012.

[22] Teodor Gabriel Crainic and Michel Toulouse. Parallel metaheuristics. In *Fleet management and logistics*, pages 205–251. Springer, 1998.

[23] Teodor Gabriel Crainic, Michel Toulouse, and Michel Gendreau. Toward a taxonomy of parallel tabu search heuristics. *INFORMS Journal on Computing*, 9(1):61–72, 1997.

[24] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, 1:1–28, 2006.

[25] C Cuda. Programming guide, 2012.

[26] Van-Dat Cung, Simone L Martins, Celso C Ribeiro, and Catherine Roucairol. Strategies for the parallel implementation of metaheuristics. In *Essays and surveys in metaheuristics*, pages 263–308. Springer, 2002.

[27] Andreas Klinkert Dinh-Nguyen Pham. Surgical case scheduling as a generalized job shop scheduling problem. *European Journal of Operational Research*, 185(3):1011–1025, 2008.

[28] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[29] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[30] Henry Fisher. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling*, pages 225–251, 1963.

[31] Marta Flamini and Dario Pacciarelli. Real time management of a metro rail terminus. *European Journal of Operational Research*, 189(3):746–761, 2008.

[32] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[33] Michael R Garey and David S Johnson. A guide to the theory of np-completeness. *WH Freemann, New York*, 70, 1979.

[34] Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[35] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations research*, 42(6):1042–1066, 1994.

[36] Paul C Gilmore and Ralph E Gomory. Sequencing a one state-variable machine: A solvable case of the traveling salesman problem. *Operations research*, 12(5):655–679, 1964.

[37] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[38] Fred Glover. Tabu search—part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[39] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[40] David E Goldberg, Jon Richardson, et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[41] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[42] Heinz Gröflin and Andreas Klinkert. A new neighborhood and tabu search for the blocking job shop. *Discrete Applied Mathematics*, 157(17):3643–3655, 2009.

[43] Heinz Gröflin, Dinh Nguyen Pham, and Reinhard Bürgy. The flexible blocking job shop with transfer and set-up times. *Journal of combinatorial optimization*, 22(2):121–144, 2011.

[44] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[45] Nicholas G Hall and Chelliah Sriskandarajah. A survey of machine scheduling problems with blocking and no-wait in process. *Operations research*, 44(3):510–525, 1996.

[46] Roger W Hockney and Chris R Jesshope. *Parallel Computers 2: architecture, programming and algorithms*, volume 2. CRC Press, 1988.

[47] James R Jackson. An extension of johnson's results on job idt scheduling. *Naval Research Logistics Quarterly*, 3(3):201–203, 1956.

[48] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[49] AHG Rinnooy Kan. *Machine scheduling problems: classification, complexity and computations*. Springer Science & Business Media, 2012.

[50] Mohamed Esseghir Lalami and Didier El-Baz. Gpu implementation of the branch and bound method for knapsack problems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1769–1777. IEEE, 2012.

[51] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[52] S Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration*, 1984.

[53] Jan K Lenstra and AHG Rinnooy Kan. Computational complexity of discrete optimization problems. In *Annals of Discrete Mathematics*, volume 4, pages 121–140. Elsevier, 1979.

[54] Alessandro Mascis and Dario Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3):498–517, 2002.

[55] Yazid Mati and Xiaolan Xie. Multiresource shop scheduling with resource flexibility and blocking. *IEEE transactions on automation science and engineering*, 8(1):175–189, 2011.

[56] Yazid Mati, Nidhal Rezg, and Xiaolan Xie. A taboo search approach for deadlock-free scheduling of automated manufacturing systems. *Journal of Intelligent Manufacturing*, 12(5-6):535–552, 2001.

[57] Nouredine Melab. *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul*. PhD thesis, 2005.

[58] Nouredine Melab, Imen Chakroun, and Ahcène Bendjoudi. Graphics processing unit-accelerated bounding for branch-and-bound applied to a permutation problem using data access optimization. *Concurrency and Computation: Practice and Experience*, 26(16):2667–2683, 2014.

[59] Nouredine Melab, Jan Gmys, Mohand Mezmaz, and Daniel Tuyttens. Multi-core versus many-core computing for many-task branch-and-bound applied to big optimization problems. *Future Generation Computer Systems*, 2017.

[60] Carlo Meloni, Dario Pacciarelli, and Marco Pranzo. A rollout metaheuristic for job shop scheduling problems. *Annals of Operations Research*, 131(1-4):215–235, 2004.

[61] Angelo Oddi, Riccardo Rasconi, Amedeo Cesta, and Stephen F Smith. Iterative improvement algorithms for the blocking job shop. In *ICAPS*, 2012.

[62] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[63] Marco Pranzo and Dario Pacciarelli. An iterated greedy metaheuristic for the blocking job shop scheduling problem. *Journal of Heuristics*, pages 1–25, 2013.

[64] Heinrich Riebler, Michael Lass, Robert Mittendorf, Thomas Löcke, and Christian Plessl. Efficient branch and bound on fpgas using work stealing and instance-specific designs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(3):24, 2017.

[65] Bernard Roy and B Sussmann. Les problemes d'ordonnancement avec contraintes disjonctives. *Note ds*, 9, 1964.

[66] Alexander Schrijver. Theory of integer and linear programming, 1986.

[67] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[68] Yu N Sotskov and Natalia V Shakhlevich. Np-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, 59(3):237–266, 1995.

[69] Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64 (2):278–285, 1993.

[70] Vincent T'kindt and Jean-Charles Billaut. *Multicriteria scheduling: theory, models and algorithms*. Springer Science & Business Media, 2006.

[71] Harry WJM Trienekens and A de Bruin. Towards a taxonomy of parallel branch and bound algorithms. Technical report, Erasmus School of Economics (ESE), 1992.

[72] Harry WJM Trienekens and Arie de Bruin. Towards a taxonomy of parallel branch and bound algorithms. 1992.

[73] Trong-Tuan Vu and Bilel Derbel. Parallel branch-and-bound in multi-core multi-cpu multi-gpu heterogeneous environments. *Future Generation Computer Systems*, 56:95–109, 2016.