CGM-based parallel solutions for a class of non-serial polyadic dynamic-programming problems

Jerry LACMOU ZEUTOUO

REPUBLIC OF CAMEROON
PEACE-WORK-FATHERLAND
****
UNIVERSITY OF DSCHANG
****
POST GRADUATE SCHOOL

RÉPUBLIQUE DU CAMEROUN
PAIX-TRAVAIL-PATRIE
****
UNIVERSITÉ DE DSCHANG
****
ÉCOLE DOCTORALE

DSCHANG SCHOOL OF SCIENCE AND TECHNOLOGY

FUNDAMENTAL COMPUTER SCIENCE, ENGINEERING, AND APPLICATIONS RESEARCH UNIT (URIFIA)

TOPIC :

CGM-BASED PARALLEL SOLUTIONS FOR A CLASS OF NON-SERIAL POLYADIC DYNAMIC-PROGRAMMING PROBLEMS

Thesis publicly defended for the award of a Doctorat/PhD degree in Computer Science

Option : Networks and Distributed Services

Specialty : Parallel Computing

By

LACMOU ZEUTOUO Jerry
Registration Number : CM-UDS-11SCI0267
Master of Science in Computer Science

Under the direction of

KENGNE TCHENDJI Vianney
Associate Professor

LÉLÉ Célestin
Professor

On December 21, 2022 *in front of the examination panel consisting of*

President		
TAYOU DJAMEGNI Clémentin	Professor	University of Dschang
Reporters		
LÉLÉ Célestin	Professor	University of Dschang
KENGNE TCHENDJI Vianney	Associate Professor	University of Dschang
Examinators		
TIEUDJO Daniel	Professor	University of Ngaoundere
NDOUNDAM René	Associate Professor	University of Yaounde I
DJOTIO NDIE Thomas	Associate Professor	University of Yaounde I

Dedication

To my parents :
LACMOU Jean and KUELA SEGNOU Antoinette

Acknowledgements

First of all, I would like to thank the Almighty Lord for the graces and mercies He has granted upon me throughout this work.

I have lived a fascinating adventure during these years of thesis work. It made me grow in both my professional and personal lives. I faced difficult moments that almost made me give up. But thanks to my supervisors, Pr. LÉLÉ Célestin and Pr. KENGNE TCHENDJI Vianney, who encouraged me, took care of me and gave me all the necessary materials to carry out this work, I did not give up. You have made me the man I am today. I also thank Pr. MYOUPO Jean Frédéric who took a keen regard over my work. It is through him that we were able to carry out our experimentations on the Dolphin cluster of the MatriCS platform of the University of Picardie Jules Verne.

I thank all the jury members for the honor you have bestowed upon me in judging this work. I also express my gratitude to the teaching staff of the Department of Mathematics and Computer Science. Their invaluable advice and expertise have allowed me to enrich my knowledge in many fields since I entered the University of Dschang in 2011.

What about my parents, LACMOU Jean and KUELA SEGNOU Antoinette, to whom I dedicate this work? Since I was born, you have made a lot of effort to ensure my education. You have labored and patiently waited until the completion of this thesis. I can only say thank you. I cannot forget the full support of my brothers and sisters, Olivier, Marlène, Dimitry, and Rhina. May the Almighty Lord infinitely bless the DIFO family, the FOGAING family, the SOPFOSSI family, the LOBE family, the SOUMANA family, the NIMPA family, the ESSOUNGUE family, the NGOUEGNI family, the MEBANN family, and the DJOUMESSI family.

My beloved friend, MEBANN Elodie, has always taken care of me throughout this work. I will always remember all the sacrifices and risks you took for me. My fan club did not abandon me and did not stop encouraging me : NGUEMKAP Romuald, GUIFO Yvan, DJOUMESSI Kerol, NOUKELA Christian, TESSA Xavier, DJOUMESSI Gina, NANKEM Ida, and LAMGOAH Marie-Noël.

I thank my research team. Although we are in different fields, we were able to make ourselves understood to each other. My elders, Dr. YANKAM Yannick and Dr. NKONJOH Armel, have contributed a lot to the completion of this work. The same goes for my cadets MVAH Fabrice, BOGNING Hermann, KAMGA Ingrid, TESSA Colette, KOMBOU Carole, and DJEUFACK Vadèle. The glances I have had at your respective works have significantly enhanced the quality of mine.

Thanks to the board of the Koossery Technology company, especially to Mr. BAKENEGHE Jean-Claude. Without him, this work would be hopeless because he allowed me to survive financially. He taught me rigor in professional work, which is obviously reflected in this thesis.

I will be eternally grateful to all the people who contributed in any way to the writing of this thesis, especially to all those who have proofread this document.

Table of Contents

Abstract

We are interested in the parallelization of a class of non-serial polyadic dynamic-programming problems in this thesis. These problems are characterized by a strong dependency between subproblems. To design efficient and portable parallel solutions to solve these problems, the CGM (coarse-grained multicomputer) model is the suitable choice because of its simplicity and its compatibility with most supercomputers.
A CGM-based parallel algorithm is a succession of computation and communication rounds. The solutions proposed in the literature give the end-user the possibility of minimizing the number of communication rounds or balancing the load between processors because both objectives are conflicting. Moreover, their main drawback is to foster the latency time of processors, which accounts for most of the global communication time.
In this work, we propose an irregular partitioning technique of the dependency graph to tackle these conflicting objectives. It consists in subdividing the dependency graph into subgraphs (or blocks) of variable size. It ensures that the blocks of the first steps (or diagonals) are of large sizes to minimize the number of communication rounds. Thereafter, it decreases these sizes along the diagonals to increase the number of blocks in these diagonals and allow processors to stay active as long as possible. These blocks are fairly distributed among processors to minimize their idle time and balance the load between them. Nevertheless, this strategy induces a high latency time of processors. Indeed, varying the blocks' sizes does not enable them to start evaluating some blocks as soon as the data they need are available. To get over this shortcoming, we propose strategies to evaluate a block as a sequence of computation and communication steps of a set of small-size blocks. The experimental results obtained show a significant performance gain compared to the most efficient solutions proposed in the literature.

Keywords: Dynamic Programming, Parallel Algorithm, CGM Model, Dependency Graph, Irregular Partitioning

Solutions parallèles sur le modèle CGM pour une classe de problèmes de programmation dynamique de type polyadique non-serial

Résumé

Nous nous intéressons à la parallélisation d'une classe de problèmes de programmation dynamique de type polyadique non-sérial dans cette thèse. Ces problèmes se caractérisent par une forte dépendance entre les sous-problèmes. Pour concevoir des solutions parallèles efficaces et portables à la classe de problèmes sus-citée, le modèle de calcul CGM (coarse-grained multicomputer) est le choix idéal en raison de sa simplicité et de sa compatibilité avec la plupart des superordinateurs.
Un algorithme CGM est une succession de rondes de calcul et de communication. Les solutions proposées dans la littérature donnent à l'utilisateur final le choix de minimiser le nombre de rondes de communication ou d'équilibrer la charge des calculs entre les processeurs car ces objectifs sont contradictoires. De plus, leur défaut majeur est de favoriser le temps de latence des processeurs, qui représente la plus grande partie du temps de communication global.
Dans ce travail, nous proposons une technique de partitionnement irrégulier du graphe de dépendances pour apporter une solution à ces objectifs contradictoires. Elle consiste à diviser le graphe de dépendances en sous-graphes (ou blocs) de tailles variables. Elle assure que les blocs des premières étapes (ou diagonales) sont de grandes tailles pour minimiser le nombre de rondes de communication. Ensuite, elle réduit leurs tailles le long des diagonales pour augmenter le nombre de blocs dans ces diagonales et permettre aux processeurs de rester actifs le plus longtemps possible. Ces blocs sont distribués de manière équitable sur les processeurs pour minimiser leur temps d'inactivité et équilibrer leurs charges de calcul. Cependant, cette stratégie induit un temps de latence élevé des processeurs. En effet, la variation de la taille des blocs ne leur permet pas de commencer l'évaluation de certains blocs aussitôt que les données dont ils ont besoin sont disponibles. Pour résoudre ce problème, nous proposons des stratégies consistant à évaluer un bloc comme une succession d'étapes de calcul et de communication d'un ensemble de blocs de petite taille. Les résultats expérimentaux obtenus montrent un gain de performance significatif comparé aux meilleures solutions proposées dans la littérature.

Mots clés: Programmation Dynamique, Algorithme Parallèle, Modèle CGM, Graphe de Dépendances, Partitionnement Irrégulier.

List of Acronyms

API	Application Programming Interface;
BSP	Bulk Synchronuous Parallel;
CGM	Coarse-Grained Multicomputer;
CPU	Central Processing Unit;
DAG	Directed Acyclic Graph;
DP	Dynamic Programming;
DSP	Digital Signal Processor;
GPU	Graphics Processing Unit;
MCOP	Matrix Chain Ordering Problem;
MPI	Message Passing Interface;
MPP	Minimum Cost Parenthesizing Problem;
OBST	Optimal Binary Search Tree;
OpenMP	Open Multi-Processing;
PRAM	Parallel Random Access Memory;
PU	Processing Unit;
TCP	Triangulation of a Convex Polygon;
VLSI	Very-Large-Scale Integration.

List of Tables

List of Figures

List of Algorithms

General Introduction

CONTENTS

Background

Dynamic programming

Over the last decades, the technological evolution led engineers to confront increasingly complex computational problems in several fields, including artificial intelligence, software engineering, image processing, operations research, economics, and electronics (Collette and Siarry, 2002). Optimization problems, consisting in optimizing (minimizing or maximizing) an objective function from input data, are frequently encountered by engineers. The single-pair shortest path problem is a well-known example. It consists in finding the shortest path between a single pair of vertices in a graph such that the sum of the weights of this path's edges is the smallest. Finding the best solution among a finite and discrete set of feasible solutions is called *combinatorial optimization*. Solving such problems is straightforward because we just need to enumerate all the possible combinations, test them one by one, and choose the best one. Nevertheless, the computational complexity theory shows that this kind of solution is not practical because its time and space complexity is exponential (Cormen et al., 2009). Several techniques have been designed to solve combinatory optimization problems in polynomial time; one of them is dynamic programming, which is widely studied and used in the literature.

Developed by Bellman Ford in the 1950s, dynamic programming (DP) consists in subdividing a problem into a set of subproblems, organizing their evaluations in such a way that each of them is evaluated just once, and combining their solutions to obtain the optimal solution (Bellman, 1957; Cormen et al., 2009; Lew and Mauch, 2007). The main goal of DP is to avoid recomputing the solutions of subproblems during the solving process of a given problem. Dependencies between subproblems are commonly illustrated by a multi-level directed acyclic graph (DAG), called *task graph* or *dependency graph*. Wah and Li (1988) presented four groups of dynamic-programming problems[1] based on the nature of the dependency of subproblems and the number of recurrence terms of the objective function: serial monadic, serial polyadic, non-serial monadic, and non-serial polyadic. The latter is characterized by a strong dependency between subproblems, which can lead to an irregular computational load between them. In this thesis, we are interested in a class of non-serial polyadic dynamic-programming problems, in particular the minimum cost parenthesizing problem (MPP), the matrix chain ordering problem (MCOP), the triangulation of a convex polygon (TCP) problem, and the optimal binary search tree (OBST) problem.

These problems can be solved by the sequential algorithm of Godbole (1973). It is sufficient to change the definition of the objective function and the input data according to the semantics of the studied problem. This algorithm runs in $O\left(n^3\right)$ time and $O\left(n^2\right)$ space by evaluating each of $\Theta\left(n^2\right)$ subproblems in $O(n)$ time. However, for the MCOP, the TCP problem, and the OBST problem, there are some speedup sequential algorithms that significantly reduce the execution time and require $O\left(n^2\right)$ time and space (Knuth, 1971; Yao, 1982). On the one hand, to solve the OBST problem, the algorithm of Knuth (1971) leverages the table used to store the indices of optimal subproblems to evaluate the $\Theta\left(n^2\right)$ subproblems in constant time. On the other hand, the algorithm of Yao (1982) solves the MCOP and the TCP problem[2] by narrowing down the number of subproblems to $\Theta(n)$ through the quadrangle inequality and evaluating each of them in $O(n)$ time.

Parallel computing : the master idea

A few decades ago, general-purpose computers were fitted with single-core processor chips. Nowadays, they are equipped with one or more multi-core processor chips and a large amount of memory. A core is a processing unit capable of per-

1. A dynamic-programming problem is a problem that can be solved through the dynamic-programming technique (Kengne, 2014).
2. There exists a one-to-one correspondence between the MCOP and the TCP problem (Hu and Shing, 1982, 1984). Thus, a solution to one of them remains relevant to the other (Yao, 1982).

forming computation operations concurrently (Pacheco and Malensek, 2021). In spite of this technological leap in hardware, the sequential algorithms of Godbole (1973), Knuth (1971), and Yao (1982) remain inefficient as the input data size increases. The reason is simple : a sequential algorithm only runs on one processing unit at a time on a computer, even if that computer has thirty-two, sixty-four, or one hundred twenty-eight processing units. The performance of such an algorithm will depend on the power of a processing unit, that is, the number of operations it is able to perform in one second.

The master idea of parallel computing is to simultaneously use several processing units to solve a problem faster by means of a parallel algorithm. Such an algorithm enables solving a problem in such a way that each processing unit works on a part of the problem at the same time and cooperates with others to solve the problem more quickly. It was possible to reduce the execution time of sequential algorithms that took days, weeks, or months to complete in a few hours, minutes, or seconds. Parallel computing has emerged at a time when humanity is faced with increasingly complex problems that need to be solved in a short time, for example to simulate the reaction of new drugs on the body, to process billions of requests, to manage millions of banking transactions, and so forth (Sterling et al., 2017). The need to exploit other computer resources, such as memory to handle problems with large amounts of data (which could not be handled by a single computer) and graphics processing unit (GPU) chips to solve problems even faster, has rapidly become apparent over the years.

A parallel computer is a computer that can execute a parallel algorithm. A parallel computer architecture consists of one or more parallel computers whose processing units work together concurrently (Kengne, 2014). An architecture is said to have a shared memory when the processing units share a common memory to exchange data, and it is said to have a distributed memory when the processing units have their own memories and exchange data through a network (Raina, 1992). These parallel computer architectures are commonly called *supercomputers*.

Parallel computing model

A parallel computing model is an abstraction of the architecture's parallel computers intended to be used to formalize in a few parameters the way parallel algorithms will behave in a given architecture (Kengne, 2014). Several models have been proposed to increase the performance of parallel algorithms. The oldest and best known of them is the PRAM (Parallel Random Access Memory) model, which allows the design of parallel algorithms on shared-memory architectures (Czech,

2017; Ferreira and Morvan, 1997; JáJá, 1992; Pacheco and Malensek, 2021). Although this model is simple, it does not reflect the characteristics of real parallel computers it describes because it assumes that access to information in memory is achieved in constant time; and thus it leads to designing parallel algorithms that are efficient in theory but inefficient in practice (Kengne, 2014). Other models, such as the systolic model and the hypercube model better fit distributed-memory architectures (Czech, 2017; Kung, 1982; Kung and Leiserson, 1978; Leighton, 1992). The main drawback of these models is that they depend on a specific network topology[3]. Thus, parallel algorithms designed in these models are not portable, that is, the algorithms usually need to be modified when changing architectures to fit the new architecture (Kengne, 2014).

The bulk synchronous parallel (BSP) model provided a solution to the shortcomings of the previous models (Valiant, 1990). Indeed, it allows better reflecting the characteristics of real parallel computers by proposing the BSP parallel computer, which is an architecture composed of a set of processors interconnected by a network and in which two processors can exchange data without going through an intermediate processor. It is thus easier to design such an architecture being that we can interconnect general-purpose computers; and thus reduce the cost of designing the architecture compared to systolic and hypercube architectures. Moreover, through the BSP model, the performance of a parallel algorithm can be estimated from a few parameters (Kielmann and Gorlatch, 2011; Valiant, 1990).

The coarse-grained multicomputer (CGM) model is a simplified version of the BSP model (Dehne et al., 1993). It allows formalizing in just two parameters the performance of a parallel algorithm : the input data size n and the number of processors p. A CGM-based parallel algorithm consists in repeating successively two phases until the problem is solved:

- a computation round where processors perform local computations on their data using the best sequential algorithm, and;
- a communication round where processors exchange data through the network.

Because of its simplicity and its compatibility with most recent supercomputers, this model is suitable for designing parallel algorithms for the class of non-serial polyadic dynamic-programming problems studied in this thesis.

3. A network topology refers to the way in which a set of nodes are connected to each other (Wu and Feng, 1984).

Research problem

Of the problems we study, the MPP is the most classical and generic because it is widely used to represent a large class of non-serial polyadic DP problems. The sequential algorithm of Godbole (1973) is besides called *generic sequential algorithm* in the literature (Kechid and Myoupo, 2009; Kengne, 2014). This algorithm organizes the subproblems level by level to obtain a DAG having the form of an upper triangular matrix (see Figure 11a). Subproblems of the same level i are evaluated at step (or diagonal) i. The parallelization constraint of this problem is the unevenness of the computational load between different diagonals. In fact, due to the strong dependency between subproblems, it will have more operations to evaluate the subproblems of diagonal $(i+1)$ than those of diagonal i. This constraint will have heavy consequences on the load balancing.

Moreover, the sequential algorithms of Knuth (1971) and Yao (1982) add new additional constraints, making the parallelization of the MCOP, the TCP problem, and the OBST problem a bit more difficult. On the one hand, the sequential algorithm of Knuth (1971) does not change the form of the DAG but neither does it guarantee that subproblems of the same diagonal will have the same computational load, compared to the sequential algorithm of Godbole (1973) where subproblems of the same diagonal have the same computational load (the number of operations is the same to evaluate these subproblems). On the other hand, according to the input data, the sequential algorithm of Yao (1982) changes the form of the DAG into a forest consisting of two binary trees (see Figure 28). Indeed, two input data of the same size but different values will have two different forms of the DAG; as a result, designers will be compelled to design a CGM-based parallel solution that fits all possible forms of the DAGs corresponding to different input data.

The standard methodology for designing CGM-based parallel solutions to solve these problems is to partition the DAG into subgraphs (or blocks) of same size, then distribute fairly these blocks among processors, and finally compute them in a suitable evaluation order. Some research has investigated the parallelization of these sequential algorithms on the CGM model in the literature (Cáceres et al., 2010; Fotso et al., 2010; Higa and Stefanes, 2012; Kechid and Myoupo, 2008a, 2008b, 2009; Kengne and Myoupo, 2012; Kengne et al., 2016; Myoupo and Kengne, 2014a, 2014b). Kengne et al. (2016) highlighted the existence of a relationship between the total execution time, the load balancing, and the number of communication rounds from the performance of previous CGM-based parallel solutions. Indeed, according to the partitioning strategy and the distribution scheme strategy

used when designing these solutions, they showed that minimizing the number of communication rounds and balancing the load of processors are two conflicting objectives when the DAG is partitioned into blocks of the same size :

1 - To promote load balancing, the blocks must be small (Kechid and Myoupo, 2008a, 2008b, 2009). Thus, if each processor has one more block to evaluate than another, their load difference will also be low. However, the number of communication rounds will be high.

2 - To minimize the number of communication rounds, the blocks must be large (Kengne and Myoupo, 2012; Myoupo and Kengne, 2014a, 2014b). Thus, since there will be few blocks, the number of communication rounds will be reduced. However, the load of processors will be unbalanced.

By generalizing the ideas of the DAG partitioning and distribution scheme introduced in (Kechid and Myoupo, 2008a, 2009; Kengne and Myoupo, 2012; Myoupo and Kengne, 2014b), Kengne et al. (2016) proposed a CGM-based parallel solution that gives the end-user the choice to optimize one criterion according to their own goal, coming for example from the characteristics of parallel computers used.

The main drawback of this solution is the conflicting optimization criteria owing to the fact that the end-user cannot optimize more than one criterion. Moreover, these criteria have a significant impact on the latency time of processors, which in turn has an impact on the global communication time. Recall that the global communication time is obtained by adding up the latency time of processors and the effective transfer time of data. Indeed, when the number of communication rounds is high, excessive communication will lead to communication overhead, which will deteriorate the global communication time. On the other hand, when the load of processors is unbalanced, the evaluation of a block will take a longer time because of its larger size. So, a processor that is waiting for this block to start or continue its computation will wait longer to receive it. As a result, no matter the criterion chosen, the global communication time will decrease the performance of CGM-based parallel solutions.

Research aim

This thesis aims to design CGM-based parallel solutions that reconcile the conflicting objectives (minimizing the number of communication rounds and balancing the load of processors) to solve a class of non-serial polyadic dynamic-programming

problems. We are confident that solutions where processors evaluate both large and small blocks will reconcile this trade-off. These solutions should minimize the number of communication rounds while balancing the load of processors and reducing the global communication time. These solutions should also minimize the idle time of processors. Indeed, in prior solutions, several processors cannot be active at the same time half in the evaluation of blocks. From this point in time, there are more and more idle processors (one more after each step). However, the closer one gets to the final steps, the higher the load of blocks becomes, and therefore their evaluation requires more computation time.

Our contributions

We have proposed three contributions to achieve our goal:

- In (Kengne and Lacmou, 2019; Lacmou and Kengne, 2018), we have proposed an irregular partitioning technique to tackle the conflicting objectives. It consists in subdividing the dependency graph into blocks of variable size. It ensures that the blocks of the first diagonals are of large sizes to minimize the number of communication rounds. Thereafter, it decreases these sizes along the diagonals to increase the number of blocks in these diagonals and allow processors to stay active as long as possible. These blocks are fairly distributed among processors to minimize their idle time and balance the load between them. Our CGM-based parallel solutions using this technique minimize also the overall computation time and the latency time of processors; and thus reduce the total execution time. They require $O\left(n^3/p\right)$ execution time for the MPP and $O\left(n^2/\sqrt{p}\right)$ for the OBST problem, each with $O\left(k\sqrt{p}\right)$ communication rounds. Here, n is the input data size, p is the number of processors, and k is the number of times the size of blocks is subdivided.

- In (Lacmou et al., 2021), we have proposed a fast sequential algorithm for the MCOP and the TCP problem. It consists in organizing the evaluation of the subproblems according to their dependencies, instead of their precedence order as in previous solutions (Myoupo and Kengne, 2014a; Yao, 1982), to solve them fastly by avoiding some unnecessary computations. It requires $O(n)$ time in many cases. Experimental results performed on the MatriCS platform showed that this sequential algorithm is $\times 18.93$ faster than the sequential algorithm of Yao (1982) and $\times 5.07$ faster than the CGM-based parallel solution of Myoupo and Kengne (2014a) on thirty-two processors.

- In (Lacmou et al., 2022a, 2022b, 2022c), we have proposed strategies to reduce the latency time of processors by allowing them to start the evaluation of blocks as soon as possible. Indeed, varying the blocks' sizes does not enable processors to start evaluating small-size blocks as soon as the data they need are available, although these data are available before the end of the evaluation of large-size blocks. We have first proposed the k-block splitting technique consisting in splitting the large-size blocks into a set of smaller-size blocks called *k-blocks*. Thus, evaluating a block by a single processor will consist of computing and communicating each k-block contained in this block. Experimental results showed that this solution is better than previous ones but leads to communication overhead when k increases. That is why we have proposed the four-splitting technique consisting in splitting the large-size blocks into four small-size blocks. It avoids communication overhead and significantly reduces the latency time of processors while preserving the complexity of our solutions using the irregular partitioning technique.

Thesis outline

The remainder of this document is organized as follows:

Chapter 1 : Parallel Computing and Dynamic Programming
This chapter presents the basic concepts of parallel computing and dynamic programming (DP), and describes the general DP formulation of the studied problems.

Chapter 2 : Parallelization of the Studied Problems : State of the Art
This chapter reviews the state of the art on parallelizing the class of non-serial polyadic DP problems we study, and also presents our fast sequential algorithm which solves the MCOP and the TCP problem with experimental results obtained.

Chapter 3 : Reconciliation of the Minimization of the Number of Communication Rounds and the Load Balancing of Processors
This chapter outlines our CGM-based parallel solutions that reconcile the conflicting objectives to solve the MPP and the OBST problem.

General Conclusion
This chapter concludes this work and provides some perspectives for future works.

CHAPTER 1

Parallel Computing and Dynamic Programming

CONTENTS

1.1 - Introduction

This chapter outlines the basic concepts of parallel computing and dynamic programming. Section 1.2 introduces the main idea of high-performance computing. Section 1.3 describes the taxonomies of parallel computer architectures. Section 1.4 then explains how to design a parallel algorithm. Parallel computing models are presented in section 1.5 as well as the motivation for choosing the CGM model in this work. Sections 1.6 and 1.7 present dynamic programming and the classification of dynamic-programming formulations, respectively. Finally, Section 1.8 describes the class of dynamic-programming problems studied in this thesis.

1.2 - High-performance computing

Toward the end of the 1980s, numerous innovations including the World Wide Web, Google, Microsoft, and Linux led to the growth and creation of many fields

in aeronautical research, space research, economics, medicine, military, and so on. Information and communication technologies have been growing exponentially ever since, owing to the development of diverse materials and computer systems (Ozdamli and Ozdal, 2015). Indeed, they allow accessing, processing, analyzing, storing, or transfering information in different forms to meet a user's need.

In 2020, a study conducted on government-imposed lockdowns of European citizens due to the COVID-19 pandemic showed that the volume of Internet traffic increased by 15-20% almost within a week because the Internet traffic demands of residential users, especially for remote working, entertainment, commerce, and education, increased dramatically (Feldmann et al., 2020). In addition, another study carried out by Li et al. (2019) on the blockchain revealed that fluctuations in the price of Bitcoin have a significant impact on the global computing power of the system. Indeed, a high Bitcoin price will attract more computing power to join the system, while a low Bitcoin price will force computing power to leave the system.

It is therefore virtually impossible to process this massive amount of data in real time with a conventional computer, although nowadays computers are equipped with multi-core processor chips. A multi-core processor is a processor made up of two or more independent processing units, called *cores*, that perform tasks concurrently (Pacheco and Malensek, 2021). Pooling hundreds, thousands, or millions of processing units and making them cooperate to process massive amounts of data and solve complex problems is the master idea of high-performance computing (Sterling et al., 2017). This kind of computer is typically called *a supercomputer*. It has been at the heart of the greatest scientific revolutions and discoveries in the last decades. There are three major types of supercomputers :

- The massive parallel processor, which is a single computer composed of a large number of processors interconnected through a high-speed network. These processors are independent, that is, they do not share memory, and usually each processor runs its own instance of an operating system (Loshin, 2013). Despite the fact that 7.8% of the world's 500 most powerful supercomputers are massive parallel processors in November 2021 according to the TOP500 ranking[1], Fugaku supercomputer, the first on the list, is a massive parallel processor. It achieved a benchmark score of 442 Pflop/s with 7630848 cores.

- The cluster, which is a set of standalone computers interconnected by a network (Kengne, 2014). According to the TOP500 ranking, 92.2% of super-

1. https://www.top500.org

computers are clusters. Made up of 4356 computers (for a total of 2414592 cores) interconnected by a dual-rail Mellanox EDR Infiniband, Summit is the second most powerful supercomputer in the world with a performance of 148.8 Pflop/s, which is three times less powerful than the first.

- The quantum computer, which uses properties of quantum physics to perform computations. They aim to solve extremely complex problems the world's most powerful supercomputers cannot and will never solve (Steffen et al., 2011). This technology gained momentum since the first quantum computer was commercialized by D-Wave in 2011, because since then, Google, Intel, IBM, and Microsoft are on a hunt for quantum supremacy[2].

1.3 - Taxonomies of parallel computer architectures

Parallel computing consists in solving a problem by using several processing units simultaneously. Each processing unit works on one part of the problem at the same time and cooperates with others to solve a problem more quickly. A parallel computer is a computer that can perform computation in parallel. A parallel computer architecture consists of one or more parallel computers whose processing units work together concurrently. Nowadays, general-purpose computers are parallel computers since they usually have one or more multi-core processor chips and sophisticated graphics processing unit (GPU) chips. The aspects that distinguish parallel computers have a strong consequence on the design of the parallel algorithms that will be executed on them. A classification according to some criteria, such as the number of instruction streams and data streams, the memory of parallel computers, the network topology, and the granularity, allows highlighting these different aspects.

1.3.1 - Classification according to the number of instruction streams and data streams

The first classification relates to the computation activity of the architecture's processing units according to the number of instruction streams and data streams being handled. Recall that the instruction stream is the sequence of instructions executed by a processing unit and the data stream is the sequence of data requested by the

2. The quantum supremacy is the goal of demonstrating that a quantum computer can solve a problem with such overwhelming speedup that no conventional supercomputer can solve the same problem in a reasonable amount of time (Zhong et al., 2020).

instruction stream (Schmidt et al., 2017). Flynn (1966) distinguishes four classes of architecture :

SISD (Single Instruction stream, Single Data stream) where a single-core processor executes a single instruction stream that operates on a single data stream (see Figure 1a). This architecture refers to the traditional von Neumann architecture.

SIMD (Single Instruction stream, Multiple Data stream) where many processing units, being part of one or more processors, execute a single instruction stream that operates on different data streams simultaneously (see Figure 1b). This architecture is suitable for image processing and matrix or vector operations, and includes some processor chips like vector processors, array processors, and GPUs.

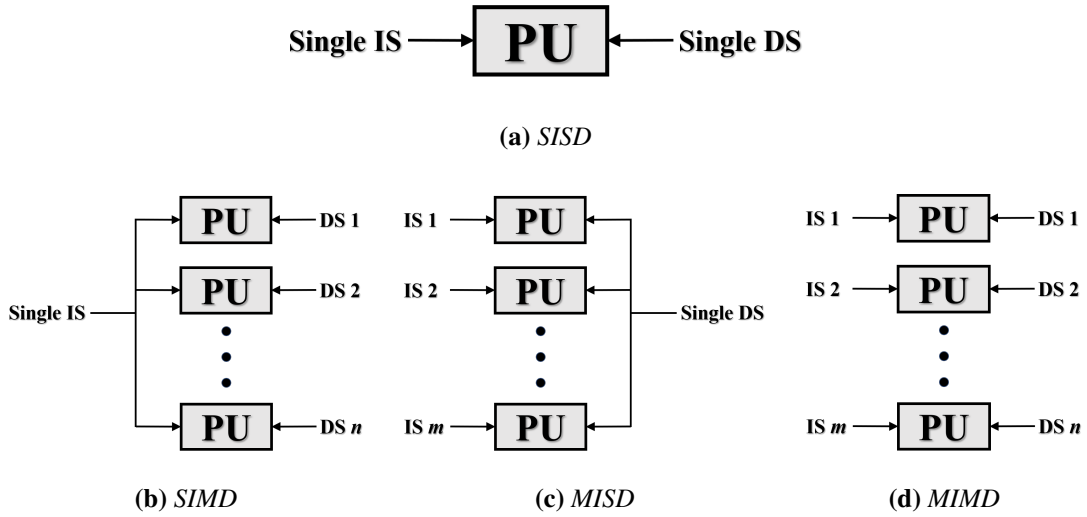
MISD (Multiple Instruction stream, Single Data stream) where many processing units execute different instruction streams that operate on a single data stream (see Figure 1c). This architecture is not very common and is sometimes used to provide fault tolerance with heterogeneous systems running on the same data to provide independent results that are compared with each other (Sitaram and Manjunath, 2012).

MIMD (Multiple Instruction stream, Multiple Data stream) where many processing units execute different instruction streams that operates on different data streams (see Figure 1d). Each processing unit has a separate instruction stream and data stream. This architecture is the most common since most modern supercomputers fall into this class.

1.3.2 - Classification according to the memory

Based on the address space and physical memory organization of the architecture's processing units, which can be shared or distributed, Raina (1992) proposes three broad classes of architecture:

SASM (Single Address space, Shared Memory) which refers to shared-memory architectures. The processing units of a multi-core computer share a common memory through an interconnection network (for example a high-speed memory bus or a crossbar switch, see Figure 2a). A single operating system ensures communication between tasks running on different processing units by writing to and reading from the global memory. There are three



(a) *SISD*

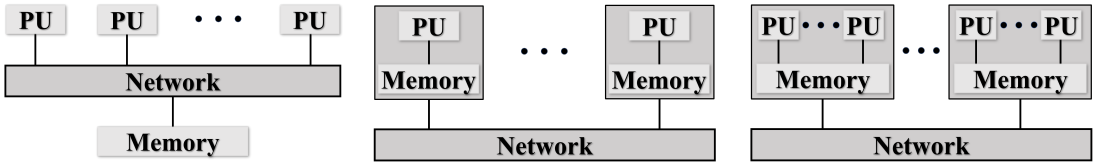
(b) *SIMD* **(c)** *MISD* **(d)** *MIMD*

Figure 1 – *Taxonomy of Flynn (1966). IS and DS stand for instruction stream and data steam respectively*

subcategories of shared memory architectures depending on memory access (Kengne, 2014):

- uniform memory access (UMA): where the shared memory is accessed by all processing units through an interconnection network in the same way that a single processor accesses its memory. Thus, they have equal access time to any memory location.
- non-uniform memory access (NUMA) : each processing unit has a portion of shared memory that has a unique address space. Any processing unit can directly access any memory location using its real address. Therefore, memory access time depends on the distance between the processing units.
- cache-only memory architecture (COMA): which is similar to NUMA except that the shared memory is composed of a cache, so that a data has no proprietary processing unit or specific location in the memory.

DADM (Distributed Address space, Distributed Memory) where several computers are interconnected by a network (see Figure 2b). Each computer has a single-core processor chip, its own private memory, and its own operating system. Each processor performs operations on its local memory and if it needs data from other processors, a communication must be established between them to exchange data. This architecture corresponds to distributed-memory architectures.



(a) *SASM* **(b)** *DADM* **(c)** *SADM*

Figure 2 – *Taxonomy of Raina (1992)*

SADM (Single Address space, Distributed Memory) which combines the features of both previous classes (see Figure 2c). This architecture corresponds to distributed shared memory architectures. It is the most used by supercomputers because of its scalability.

1.3.3 - Classification according to the network topology

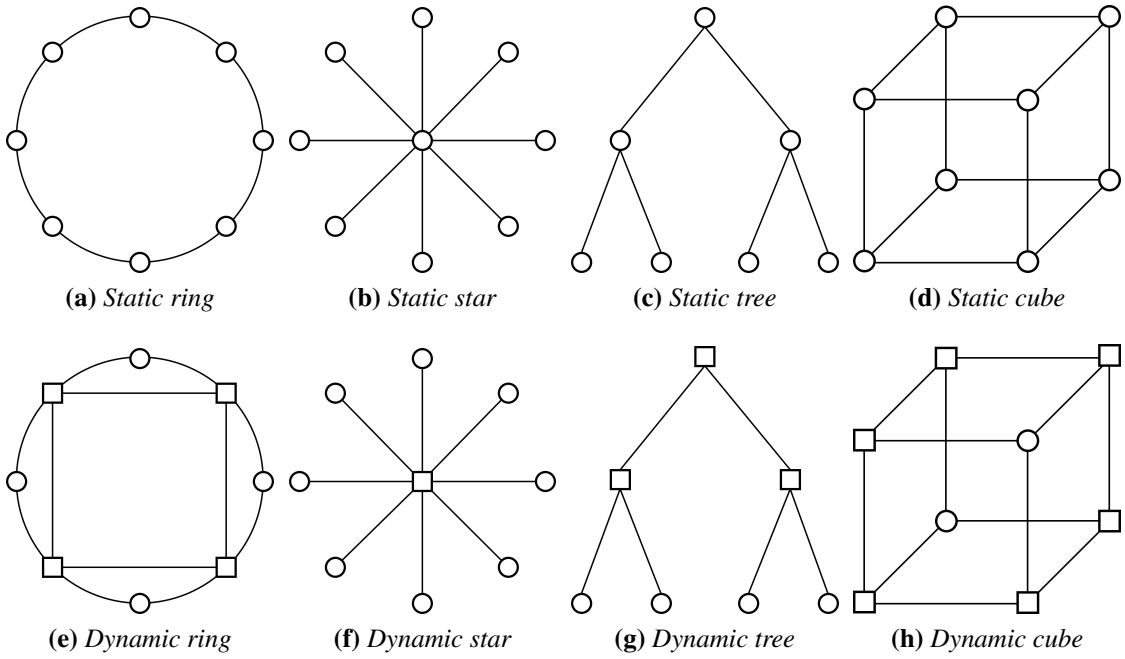
As seen earlier, the parallel computers of an architecture must be interconnected by a fast and reliable network to ensure efficient communication between them. The way they are connected by links (wire or fiber) and switches have a significant impact on the cost, applicability, scalability and performance of the architecture. These networks can be divided into two categories :

Static networks where computers are connected by point-to-point links (see Figures 3a, 3b, 3c, and 3d). The topology defined during the design of the architecture does not change. Thus, each computer knows its neighbors as in Figure 3a, where each computer has two neighbors. If a computer sends a data to a non-neighboring computer, it must absolutely transit through all the computers separating them along a given path (Kengne, 2014).

Dynamic networks where each computer is connected to a network of switches over one or more links (see Figures 3e, 3f, 3g, and 3h). Thus, the topology can be dynamically updated according to the application requirements running on the architecture. When a data wants to be sent, switches find optimal paths between computers.

1.3.4 - Classification according to the granularity

In parallel computation, granularity is the amount of computation related to a task that has been assigned to a processing unit between communication and/or synchronization phases (Grama et al., 2003). It is also defined as the measure of the



(a) *Static ring* **(b)** *Static star* **(c)** *Static tree* **(d)** *Static cube*

(e) *Dynamic ring* **(f)** *Dynamic star* **(g)** *Dynamic tree* **(h)** *Dynamic cube*

Figure 3 – *Static and dynamic network topologies. The circle nodes are parallel computers and the square nodes are switches. A link connects either two computers, two switches, or a computer and a switch*

ratio of computation to communication (Hwang, 2008). The grain size of a task depends on the type of parallelism that must be performed on the architecture. It has consequences on the performance of a parallel algorithm such as the total number of computation and communication steps, and on the load balancing. Two major types of parallelism exist depending on granularity :

Fine-grained parallelism where the computation to communication ratio is low because processing units perform small tasks. It implies high communication overhead as the number of tasks is high. However, this parallelism is suitable for architectures where communications are fast, such as shared-memory architectures, and eases the load balancing (Hwang, 2008). Indeed, the communication overhead can cause a significant drop in performance if the network is not fast.

Coarse-grained parallelism where the computation to communication ratio is high because processing units perform large tasks. Since the number of tasks is low, it entails low communication overhead (Hwang, 2008). However, this parallelism makes the load balancing difficult, especially for problems where the amount of computation of tasks is not the same although they may have the same size. In addition, it exacerbates the idleness of processing units because some can perform tasks while others are idle.

1.4 - Designing parallel algorithms

Designers of sequential algorithms do not generally care about the characteristics of the computer where these algorithms will be executed because these algorithms exploit only a single processing unit of a multi-core or multi-processor computer. In contrast, a parallel algorithm simultaneously exploits the resources of an architecture's parallel computers (CPU chips, GPU chips, memory, etc.) to solve a problem on a large amount of data more quickly (and thus potentially speed up the production time of a product, save money, etc.). This is why designers of parallel algorithms must know the architecture where these algorithms will be executed. Indeed, the architecture guides designers on the type of parallelization, the source of parallelism, the parallel programming model, and the parallel computing model (described in Section 1.5) to adopt.

1.4.1 - Manual parallelization versus automatic parallelization

There are two approaches to parallelizing a sequential algorithm: manual parallelization and automatic parallelization.

Manual parallelization

The purpose of manual parallelization is to give designers of parallel algorithms full control over the implementation and execution of the parallel algorithm in a given architecture. More specifically, the designers must analyze the problem to be solved, divide the problem into subproblems (or tasks), analyze the dependencies between tasks, assign the tasks to processing units, define a communication and/or synchronization scheme to allow processing units to exchange data, organize the evaluation of the different tasks until the problem is solved completely, combine the results of different processing units, etc. Of the existing methodologies for manually designing a parallel algorithm, the best known is that of Foster (1995). It is made up of four steps:

1 - Partitioning : this process decomposes into tasks the computation that will be performed to solve the whole problem and the data operated on by this computation. The tasks obtained after partitioning are intended to be executed concurrently.

2 - Communication : this process determines how tasks will communicate with each other, since the computation to be performed in one task sometimes requires data associated with another task.

3 - Agglomeration : this process goes back to the decision made in the two previous steps to group tasks into larger tasks according to the architecture's parallel computers. This step aims to improve performance and reduce the cost of developing the parallel algorithm.

4 - Mapping : this process assigns tasks to processing units in an attempt to maximize the use of processing units and minimize communication costs. The tasks are assigned either statically (i.e. determined during the implementation of the algorithm) or dynamically (i.e. determined during the execution of the algorithm) to improve load balancing.

Manual parallelization is usually a time-consuming and complex process. Designers typically encounter many errors and when they correct them, they test their algorithms on large amounts of data (these tests can take days or even weeks) to observe the performance before adopting or redesigning the proposed algorithm. In addition, designers need to have a strong background in parallel computing before designing a parallel algorithm.

Automatic parallelization

Automatic parallelization is the technique of automatically transforming a sequential algorithm into an equivalent parallel algorithm using the operating system's compiler or specific tools (Mabrouk, 2016). Indeed, it is very common to use iterative structures of nested loops in algorithms; however, these structures consume most of the execution time. It is thus on this part that the compilers ensuring the automatic parallelization will operate by enabling the parallel execution of the loop iterations on processing units. This parallelization consists in making the sequential algorithm undergo a series of legal transformations, in the sense that they preserve the semantics of the initial algorithm, by first carrying out an analysis of dependencies within this sequential algorithm, then detecting the inherent parallelism in the algorithm, and finally determining a series of transformations of this algorithm in respect with its dependencies and making it possible to extract the inherent parallelism (Mabrouk, 2016). There are several loop transformations such as loop distribution, loop interchange, loop skewing, loop fusion, and loop tiling.

Automatic parallelization aims to simplify and reduce the development time of parallel algorithms, which are significantly more difficult to design than sequential algorithms, without necessarily having knowledge of parallel computing. Although it is suitable for shared-memory architectures, it also makes parallel algorithms more portable. However, when faced with a complex sequential algorithm, some semantic information can be lost during the analysis of dependencies and can lead

to the production of an incorrect parallel algorithm or a parallel algorithm that is less efficient than the initial algorithm (Midkiff, 2012). In some cases, for example when the loops do not have the same number of iterations, automatic parallelization still requires the intervention of a designer who has a strong competence in parallel computing.

1.4.2 - Control parallelism versus data parallelism

Consider a sequential algorithm that executes m instructions on each element of an array of size n. To parallelize this algorithm, the designer must spread the tasks over the resources of the architecture's parallel computers. They have a choice between two sources of parallelism.

The first is to spread the m instructions over processing units. Thus, each processing unit will execute a part of instructions on each element of the array (i.e. on the same data or on different data). This form of parallelism is called *control parallelism*. It is widely used in shared-memory architectures. The efficiency of such a solution depends on the number of instructions to be performed on each element of the array. The most common control parallelism is called *pipelining*.

The second is to spread the n elements of the array over processing units. Each processing unit will thus execute the m instructions on each element of the subset of the array that has been assigned to it. This form of parallelism is called *data parallelism*. It is both used on shared-memory and distributed-memory architectures. The efficiency of this solution depends on the size of the array (i.e. the input data size). The advent of GPU chips has expanded the usage of this form of parallelism by designers of parallel algorithms.

It is obviously possible to combine these two forms of parallelism. In that case, each processing unit will execute part of the instructions on each element of the subset of the array that has been assigned to it. This type of solution is used in distributed shared-memory architectures (see Figure 2c). Its objective is to use the maximum resources of the architecture's parallel computers to run faster than the two basic forms.

1.4.3 - Parallel programming models

A parallel programming model is a set of program abstractions for fitting parallel activities from the application to the underlying architecture's parallel computers (Vitorović et al., 2014). It acts as an interface between the parallel algorithm and the architecture through its basic operations such as arithmetic operations, spawning of tasks, reading and writing in shared-memory architectures, or sending and

receiving messages in distributed-memory architectures (Kessler and Keller, 2007). Of the parallel programming models, shared-memory and message-passing models are the best known. The terminology commonly used in literature is as follows: tasks are carried out by threads and processes in the shared-memory programming model and the message-passing programming model, respectively (Czech, 2017; Pacheco and Malensek, 2021; Quinn, 2003).

Shared-memory programming model

The shared-memory programming model, as the name indicates, is suitable for shared-memory architectures where the processing units share a common memory space (see Figure 2a). In this model, the program corresponding to a parallel algorithm is executed by one or more threads, which are assigned equally to each processing unit by the operating system (Katagiri, 2019b). Indeed, the operating system allocates some resources to the process, including memory space to store instructions and data, the set of registers, etc. All the threads running the process share the address space of a designated area of memory allocated to this process; but they have also a few resources for their exclusive use, such as the stack, program counter register, memory to store private variables (Czech, 2017). So, they implicitly communicate by storing computation results and messages that can be read by other threads asynchronously (Czech, 2017; Vitorović et al., 2014). Mechanisms such as locks, semaphores, and monitors are used to ensure mutual exclusion.

The shared-memory programming model uses the fork-join paradigm, which corresponds to an on-demand parallelism (Czech, 2017; Quinn, 2003). At the beginning of the program execution, only one thread, called *master thread*, executes the sequential portions of the parallel algorithm. In the parts where parallel operations are required (also called *parallel region*), the master thread implicitly spawns or wakes up additional threads; this corresponds to a *fork*. These operations are spread over the master thread and the created threads, and they work simultaneously to accomplish their tasks. At the end of the execution of the parallel region, there is a synchronization barrier to allow the created threads to die or be suspended; this corresponds to a *join*. Thereafter, the master thread resumes the execution that was suspended at the time of the encounter with the parallel region (Czech, 2017; Quinn, 2003). Figure 4 gives an overview.

Nowadays, OpenMP (Open Multi-Processing)[3] is the mainstream standard for shared-memory programming. It is an application programming interface (API)

3. www.openmp.org

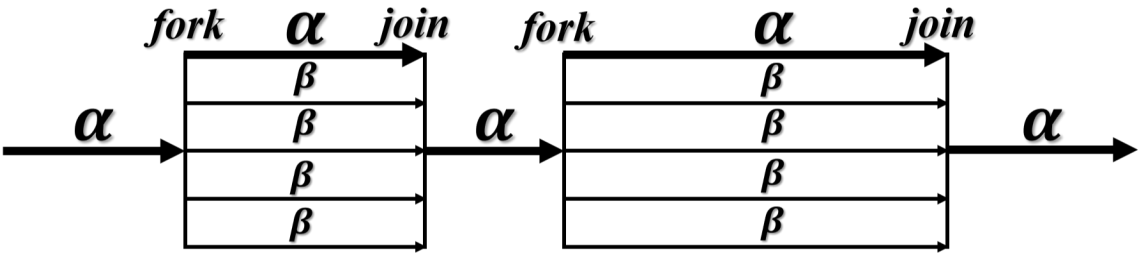


Figure 4 – *Fork-join paradigm in shared-memory programming model.* α *refers to the master thread and* β *a created thread*

that allows writing parallel algorithms in C, C++, and Fortran languages through compiler directives, library functions, and environment variables. OpenMP does not provide automatic parallelization (Czech, 2017; Katagiri, 2019b). Indeed, to design a parallel algorithm with OpenMP, designers usually have to begin with writing the sequential algorithm before turning it into a parallel algorithm. So in the first step, designers must identify the parallelism contained in a sequential algorithm by detecting parallel regions. Then, by applying directives, designers tell the compiler which sections to run in parallel, and provide information about how the computation work contained in those sections is spread over the threads. Every directive begins by the prefix `#pragma omp` followed by the directive name and the number of clauses that can be separated by commas or white spaces (Czech, 2017). The number of threads created by the master thread can be defined by the designer using the `omp_set_num_threads` function or the `OMP_NUM_THREADS` environment variable.

Message-passing programming model

In message-passing programming model, a set of processes (equally distributed on processing units) executes the same program on different data, and exchanges data explicitly by sending and receiving messages (Foster, 1995; Hager and Wellein, 2010; Levesque and Wagenbreth, 2010; Quinn, 2003). Each process has its own address space. This model targets distributed-memory architectures (see Figure 2b); nevertheless it can also be used by shared-memory architectures and distributed shared-memory architectures (see Figure 2c) (Hager and Wellein, 2010). Unlike the shared-memory programming model, where threads are created when the program first enters a parallel region and are blocked while the program executes a non-parallel region, in this model, processes are created at the beginning of the execution and are active until the program terminates (Vitorović et al., 2014). This model gives designers full responsibility for mapping data to processes, coordinating data communication and synchronization between processes (Nielsen, 2016).

MPI (Message Passing Interface) is the *de facto* standard for message-passing programming. It is a library that provides functions, beginning by the prefix `MPI_`, for various kinds of communications and for managing processes (Pacheco and Malensek, 2021). MPI is implemented by many free softwares, including MPICH[4] and OpenMPI[5], that are supported by most common programming languages such as C, C++, Java, Python, and Fortran.

Two functions are fundamental to designing a parallel algorithm using MPI (Czech, 2017; Foster, 1995). The first function, `MPI_Init`, must be called to perform some setup before any other MPI functions and must be called exactly once per process. The second function is `MPI_Finalize`. It must be called before the end of the parallel algorithm to allow the system to cleanup resources that have been allocated by MPI. No further MPI functions can be called after `MPI_Finalize`.

When a user runs this parallel algorithm, he must specify how many processes should be performed in parallel. Functions `MPI_Comm_size` and `MPI_Comm_rank` respectively determine the number of processes entered by the user and the integer identifier assigned to the current process. Indeed, processes usually identify each other by ranks in the range $0, 1, \ldots, p-1$, where p is the number of processes (Pacheco and Malensek, 2021). These ranks are used by the communication functions to send or receive a message. Two modes of communication exist according to the number of processes involved (Czarnul, 2018) :

Point-to-point communication where two processes are involved in a communication routine. If a process A sends a message to a process B, then the process A must specify the rank of the process B. When the process B needs to receive the message from the process A, the process B can specify the rank of the process A. In case the process B does not know who sent the message, it is possible to allow receiving from any process through the wildcard `MPI_ANY_SOURCE`. Point-to-point communication can be characterized depending on how synchronization is performed (Czarnul, 2018; Katagiri, 2019a):

- The communication is said to be *blocking* when the receiving function `MPI_Recv` does not return to the program after calling the corresponding sending function `MPI_Send` until the data are received and the data are entirely copied into a receiving buffer.

4. www.mpich.org
5. www.open-mpi.org

- The communication is said to be *non-blocking* when the receiving function `MPI_Irecv` returns to the program immediately even if the corresponding sending function `MPI_Send` is not called. There is also the sending function `MPI_Isend` that only starts the sending operation a message without blocking the process. The process may later use on the function `MPI_Test` to query the status of the sending operation or the function `MPI_Wait` to wait for its completion (Czech, 2017).

Collective communication where several processes are involved in a communication routine. In contrast to the point-to-point communication, all processes involved in a collective call invoke the same function. Some arguments, nevertheless, can be different depending on the rank of a process (Czarnul, 2018). The following are the characteristics of some of the functions :

- the function `MPI_Barrier` is used to synchronize processes, i.e. a process is only allowed to pass the barrier once each of the other processes has reached the barrier;
- the function `MPI_Bcast` is used in the case where one process broadcasts data to all other processes;
- the function `MPI_Alltoall` is used in the case where all processes send data to all other processes.

1.4.4 - Performance of a parallel algorithm

One of the most important tasks for designers of parallel algorithms is to measure the performance of their algorithms and compare them with those of sequential algorithms that solve the same problem. The goal is to show how the simultaneous use of several resources is more efficient than using only one to solve a problem.

Total execution time

The total execution time of a parallel algorithm is usually measured by the difference between the time the first process starts computation and the time the last process finishes the computation (Foster, 1995). This definition is not fully adapted to the shared-memory programming model since the master thread starts and ends the execution of a parallel algorithm by spawning or waking up additional threads if needed. In any case, each process (or thread) computes, communicates, or idles while the algorithm is running. Thus, the execution time is mainly composed of :

1 - The computation time, which is the time where a process or a thread performs the computations to carry out the tasks that have been assigned to it.

It depends on several factors such as the problem size, the number of processes, the number of tasks to be performed by a process, the grain size of a task, the characteristics of processing units and memory, and so on. An important challenge for designers is usually to allow processes to perform approximately the same computational load to achieve approximately the same computation time.

2 - The communication time, which is the time spent by processes or threads to respectively send and receive messages to each other, or to read and write in the shared memory. In the message-passing programming model, the communication time is often obtained by adding up the effective transfer time of the data and the latency time of processes (Kengne, 2014). Recall that the latency time is the time during which a process waits for a data to continue its computations. Commonly, intra-processor communication, where two depending tasks are located on the same processor chip, is faster than inter-processor communication, where two depending tasks are located on different processor chips. This is because inter-processor communication depends on the physical bandwidth of the communication channel linking the source and destination processors (Foster, 1995).

3 - The idle time, which is the time where a process does neither computation nor communication while running the parallel algorithm. It is usually due to a lack of computation in cases where one process performs all the tasks assigned to it (and thus completes its execution) while the others still have tasks to perform. It is also due to the operations of spawning or waking up threads, deleting or suspending threads in the shared-memory programming model. The designers of parallel algorithms seek in most cases to reduce this idle time by allowing processes to remain active, i.e. to perform computation or communication, as long as possible.

Denoting respectively by T^i_{comp}, T^i_{comm}, and T^i_{idle} the time spent on computation, communication, and idling, on the ith process, Foster (1995) expresses the total execution time T_{par} by Equation (1.1) :

$$T_{par} = T^i_{comp} + T^i_{comm} + T^i_{idle} \tag{1.1}$$

Speedup and efficiency

Speedup is the most commonly used metric for evaluating the performance gain from parallel computing. It shows how many times the execution time of a sequen-

tial algorithm, denoted T_{seq}, can be reduced by using a parallel algorithm (Czech, 2017). The speedup is defined as :

$$S = \frac{T_{seq}}{T_{par}} \tag{1.2}$$

The number of processes (or threads) p is the maximum value of the speedup, i.e. a parallel algorithm must be at most p times faster than the best sequential algorithm using p processes. However, the speedup achieved is typically less than p. The reason may be due to the fact that some portions of a sequential algorithm cannot be parallelized, to the unbalanced computational loads of processes, to the overhead associated with the communication time between processes and synchronization of their operations (Czech, 2017; Quinn, 2003).

The efficiency of a parallel algorithm is a metric that measures process utilization. It highlights the efficiency with which a parallel algorithm uses the resources of the architecture's parallel computers (Foster, 1995). It is usually expressed in percentage and defined as :

$$E = \frac{T_{seq}}{p \times T_{par}} = \frac{S}{p} \tag{1.3}$$

Scalability

It is important in analyzing the performance of a parallel algorithm to study how the performance of the algorithm varies with parameters such as the problem size and the number of processes (or threads). Let's refer to a parallel algorithm executing on the architecture's parallel computers as a parallel system. The scalability of a parallel system is a measure of its capacity to increase speedup or maintain constant efficiency with a growing number of processes used for computation (Czech, 2017; Foster, 1995; Grama et al., 2003; Quinn, 2003). In general, the efficiency of a parallel system decreases as the number of processes increases. On the other hand, when the problem size is increased while keeping the number of processing units constant, the efficiency of a parallel system increases. A good parallel system is one in which the efficiency can be kept constant as the number of processes and the problem size increase (Grama et al., 2003).

1.5 - Parallel computing models

A parallel computing model is an abstraction of an architecture's parallel computers intended to be used by designers of parallel algorithms (Kengne, 2014). It

formalizes in a few parameters the way parallel algorithms will behave in a given architecture. From the characteristics of a model, it is possible to determine the time or space complexity of a parallel algorithm without having to implement or execute it; and thus predict whether or not it will provide good performance, or compare it with other algorithms solving the same problem to determine which is the best algorithm for this problem (Kengne, 2014).

A good parallel computing model should meet the following expectations (Ferreira and Morvan, 1997; Foster, 1995; JáJá, 1992):

- a model should be simple and should not have too many parameters to facilitate its usage;
- a model must be well-defined to serve as a common platform for different designers;
- a model must reflect the characteristics of most existing architectures;
- parallel algorithms designed according to a model should not be dependent on a specific architecture;
- theoretical predictions of the performance of parallel algorithms must be in accordance with their execution on the architecture's parallel computers;
- a model must survive the evolution of parallel programming models used by parallel algorithms;
- a model must be deterministic, i.e. when running a parallel algorithm several times with a particular input on a model, it must always lead to the same output.

Unfortunately, it is difficult to design a model that meets all of the above criteria (JáJá, 1992). This section presents the parallel computing models that are most commonly used to design parallel algorithms.

1.5.1 - PRAM model

The PRAM (Parallel Random Access Memory) is undoubtedly the most popular shared-memory model in parallel computing (Czech, 2017; Ferreira and Morvan, 1997; JáJá, 1992; Pacheco and Malensek, 2021). This model assumes that a parallel computer is composed of a set of processing units and of a memory to which each processing unit has a constant access time to read and write a data. They operate synchronously under the control of a common clock during computation

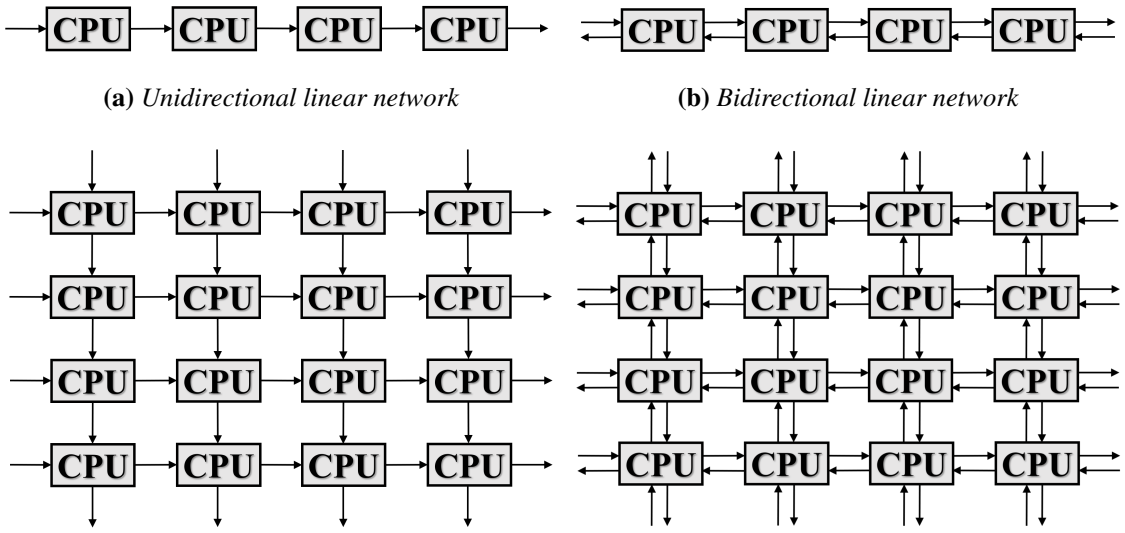
steps; hence, access conflicts to the same shared-memory location may occur. Exclusive Read (ER) means that only one processor can read from a memory location at a given time, and on the other hand, Concurrent Read (CR) means that several processors can read the contents of the same memory location. A similar classification has been proposed for write accesses. The writing operation is exclusive (EW for Exclusive Write) if only one processor can write in a memory cell at a given time, and concurrent (CW for Concurrent Write) in the case where several processors can modify the contents of the same memory cell. By combining the various possibilities of reading and writing, there are four versions of the PRAM model: ERCW, CRCW, EREW, and CREW (Czech, 2017; JáJá, 1992).

The PRAM model is simple. It allows, in most cases, to know if a problem can be parallelized or not, to design parallel algorithms without worrying about communications between processing units, and to easily analyze them (Kengne, 2014). However, this model is too abstract, i.e. it hides the low-level details of the machines it describes for ease and simplicity, to the point of hiding from designers some that are essential to the design of efficient parallel algorithms. It is far from real parallel computers because maintaining a constant access time to the shared memory for a large number of processing units is physically unfeasible and technically impossible (Czech, 2017). So, a PRAM-based parallel algorithm cannot be implemented in a real parallel computer and must be readapted to the chosen architecture's parallel computers (Ferreira and Morvan, 1997; Kengne, 2014). Moreover, a communication operation is much more expensive than a computation operation and therefore much more influential on the execution time of parallel algorithms in real parallel computers. Thus, hiding this detail from designers leads to the design of inefficient parallel algorithms (Kengne, 2014).

1.5.2 - Systolic model

Introduced by Kung and Leiserson (1978), the systolic model has proven to be a powerful tool for the design of specialized embedded processors. A systolic architecture is a network of processors that perform simple operations and exchange data regularly (Kung, 1982). Indeed, systolic-based parallel algorithms operate synchronously, that is, at each time unit, a processor receives data from some neighbors, then performs local computations, and finally sends data to some of its neighbors (JáJá, 1992).

To describe a systolic architecture, it is necessary to specify many details including the network topology of processors, the description of processors, the program that processors will execute, and the data stream consumed by the network to



(a) *Unidirectional linear network* **(b)** *Bidirectional linear network*

(c) *Unidirectional orthogonal network* **(d)** *Bidirectional orthogonal network*

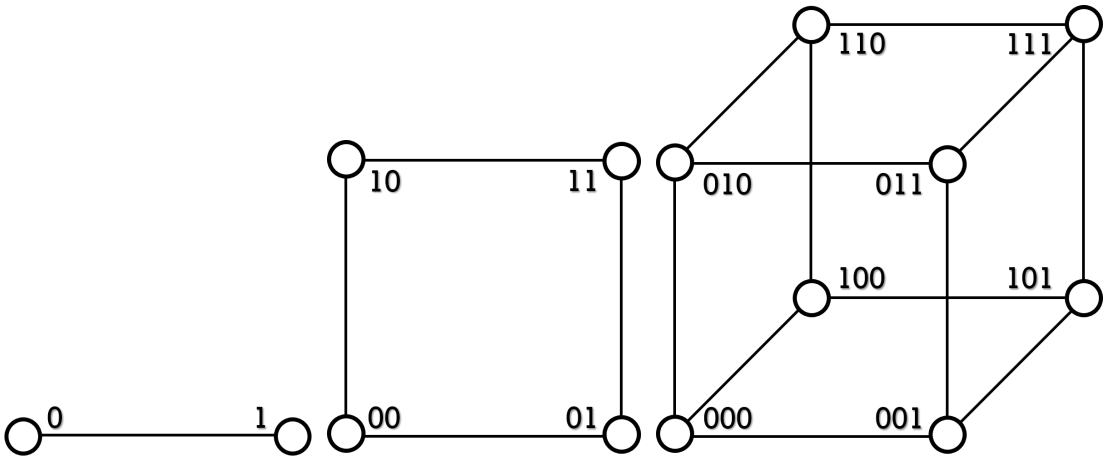
Figure 5 – *Network topologies of processors in systolic architectures*

produce a solution. Figures 5a, 5b, 5c, and 5d illustrate some network topologies of processors in systolic architectures.

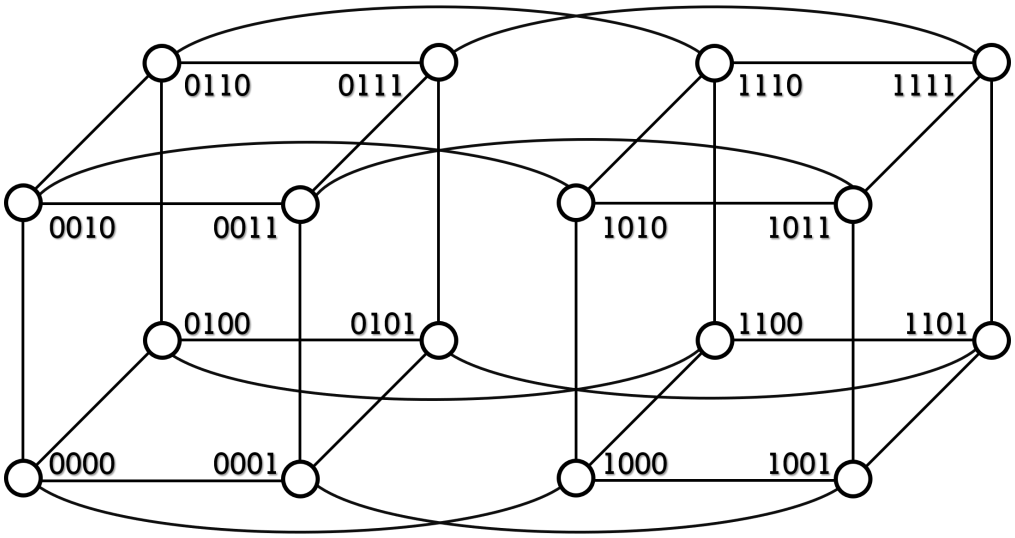
The systolic model owes its success to its achievement toward the demand for extremely fast, low-cost supercomputers, and the significant reduction in execution time for more sequential algorithms such as matrix-vector multiplication, matrix-matrix multiplication, matrix inversion, fast Fourier transform, and LU decomposition. This has led to the realization of specialized embedded processors dedicated to many applications like image and signal processing (Kung, 1982). However, this model is too realistic, i.e. it provides designers with the tiniest details (network topologies, routing techniques, etc.) of the architecture's parallel computers it describes in an attempt to increase the efficiency of parallel algorithms, to the point where these details become intrinsic to the validity and efficiency of the parallel algorithms developed. Thus, running these algorithms on computers other than those described by this model will greatly degrade their efficiency because they depend heavily on the chosen network topology; consequently, it is often required to change the parallel algorithm when the network topology evolves (Kengne, 2014).

1.5.3 - Hypercube model

A hypercube architecture is composed of $p = 2^d$ processors, indexed from 0 to $p-1$, that are interconnected within a d-dimensional cube (Czech, 2017; Ferreira and Morvan, 1997; JáJá, 1992; Leighton, 1992). The latter is a connected graph consisting of 2^d nodes and $d \times 2^{d-1}$ edges. Each node corresponds to a d-bit binary



(a) *One-dimensional cube* **(b)** *Two-dimensional cube* **(c)** *Three-dimensional cube*



(d) *Four-dimensional cube*

Figure 6 – *Network topologies of processors in hypercube architectures*

string, and two nodes are linked with an edge if and only if their binary strings differ only at one position (Czech, 2017; Leighton, 1992). Figures 6a, 6b, 6c, and 6d respectively depict a one-dimensional, two-dimensional, three-dimensional, and four-dimensional cube. It can be noticed that 2^d processors are needed to extend a d-dimensional cube into a $(d+1)$-dimensional cube.

As in the case of the PRAM model and the systolic model, the processors work synchronously. At each step, each processor can send data to one of its neighbors, receive data from one of its neighbors, and perform local computation on that data. A message communicated between two non-neighboring processors is routed through intermediate nodes according to a specific routing algorithm (Mabrouk, 2016). The hypercube model is popular because of its regularity, its versatility, its many interesting graph-theoretic properties, and its ability to handle many com-

putations quickly and simply. However, the logarithmic growth of the number of connections of each processor with the size of the network is a main drawback of this model (JáJá, 1992; Leighton, 1992). Moreover, like in the systolic model, hypercube-based parallel algorithms are not portable (Kengne, 2014).

1.5.4 - BSP model

Valiant (1990) proposed the bulk synchronous parallel (BSP) model as a solution to the efficiency-portability trade-off caused on the one hand by models that are too abstract like the PRAM model, and on the other hand by models that are too realistic like the systolic and hypercube models. Indeed, the idea of this model comes from the observation made on the von Neumann (1945) model, which built a *bridge* between computer designers and sequential algorithm designers. Von Neumann's model allows each of them to focus on their areas of expertise to evolve their different technologies (processors, storage memory, networks, programming languages, compilers, and so on) without worrying about compatibility problems. In the same vein, Valiant (1990) proposed the BSP parallel computer, which is an architecture of parallel computers composed of a set of processors equipped with memory, an interconnection network allowing point-to-point communications between processors, and a synchronization mechanism allowing a barrier synchronization accross all processors. The abstraction made on the characteristics of the interconnection network and on the relationship of the memories with processors allows the BSP model to encompass all shared-memory architectures and all distributed-memory architectures; and thus to reflect almost all supercomputers (Kengne, 2014). This is the reason why this model, like the von Neumann model, is widely used till nowadays. It served as a core model for the development of other models such as the LogP model (Culler et al., 1993), the CGM model (Dehne et al., 1993), and the Multi-BSP model (Valiant, 2011); and the development of libraries such as the BSPlib library (Hill et al., 1998), the Green BSP library (Goudreau et al., 1999), the MulticoreBSP library (Yzelman and Roose, 2014).

A BSP-based parallel algorithm consists of a succession of supersteps. As shown in Figure 7, a superstep has three phases (Kielmann and Gorlatch, 2011; Valiant, 1990):

1 - A local computation phase where each processor asynchronously performs computation operations by using the best sequential algorithm that solves the problem sequentially, on data in the processor's local memory. These data are put into the local memory either at beginning of the execution of the parallel algorithm or by communication operations of previous supersteps.

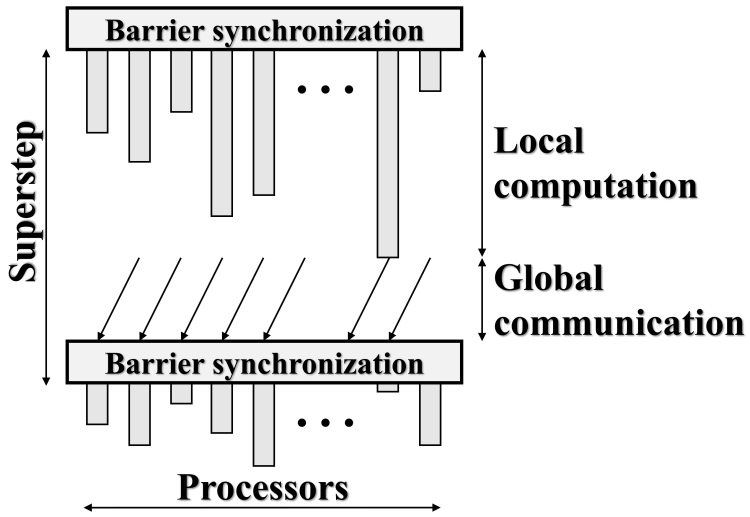


Figure 7 – *Description of a BSP superstep*

2 - A global communication phase to exchange data between processors. A communication operation does not become effective before the next superstep begins, i.e. the receiver cannot use received data until the current superstep is finished.

3 - A synchronization phase to finalize communication operations and enable access to received data by the receiving processors before any operation of the next superstep begins.

The performance of a BSP-based parallel algorithm is expressed using three parameters p, l, and g; where p is the number of processors, l is the time required for a barrier synchronization, and g is the time needed for transporting a memory word between two processors (Kielmann and Gorlatch, 2011; McColl, 1995; Valiant, 1990). All communication operations made during a superstep are considered together in what is called an *h-relation*. It is a global communication scheme defined on a set of processors where each one can send or receive at most h words (Kengne, 2014). The execution time of a BSP-based parallel algorithm is the sum of the execution times of all supersteps. The execution time of a superstep is obtained by adding up the maximum local computation times of all processors (denoted by w), the global communication time, and the synchronization time :

$$T_{superstep} = w + g \times h + l \tag{1.4}$$

1.5.5 - CGM model and motivation behind the choice of this model

Introduced by Dehne et al. (1993) to parallelize sequential algorithms for geometric problems, the coarse-grained multicomputer (CGM) model is a simplified version of the BSP model. It gets rid of the parameters l, g, w, and h of the BSP model and retains only the parameters n and p, which are the input data size and the number of processors, respectively. To represent recent supercomputers composed of several thousand of processors that can process millions or billions of data, this model considers that p must be significantly smaller than n ($p \ll n$). Unlike the BSP model, this model captures the intrinsic computation and communication features of real parallel computers at two levels:

At the processors' memory level: parallel computers of today's architectures are equipped with *state-of-the-art* processors connected to an interconnection network. The CGM model considers that each of the p processors is endowed with a sizable amount of local memory so that they can store $O(n/p)$ input data (Olariu, 2008). This is the reason why this model is coarse-grained (Dehne et al., 1993; Ferreira and Morvan, 1997).

At the communications level: conventional networks allow the transfer of large amounts of data. The CGM model exploits this feature by allowing processors to send and receive at most $O(n/p)$ data. Indeed, all the information sent from a given processor to another processor during a communication cycle is gathered in a single long message, thereby minimizing the overall message overhead (Dehne et al., 2002). Moreover, during communication operations, synchronization is done implicitly, unlike the BSP model where synchronization is explicitly part of a superstep (Lassous et al., 2000).

The CGM model inherits the portable feature of BSP-based parallel algorithms. Since most personal computers are parallel computers nowadays, the cost of designing a CGM-compatible architecture is relatively low and easy to build. All of these make this model the ideal choice for designing parallel algorithms in this thesis.

A CGM-based parallel algorithm consists of alternating local computation and global communication rounds, a pair of which corresponding to a BSP superstep (Dehne et al., 2002). In each computation round, a processor typically processes its data locally by using the best sequential algorithm to minimize the local computation time per processor. In each communication round, the total data exchanged by each processor are limited to $O(n/p)$. Designers of parallel algorithms will no longer seek to minimize the overall amount of data exchanged, as in the BSP

model, but rather design algorithms with a small number of communication rounds that only depends on p and not on n (Ferreira, 2001; Lassous et al., 2000). Indeed, for most parallel algorithms based on previous models, the number of communication rounds and the resulting message overhead dramatically increase when the input data size and the number of processors grow, resulting in a considerable performance loss. CGM-based parallel algorithms avoid these problems because they have only a small number of communication rounds whose message size grows with the input data size, which greatly improves performance and scalability (Chan et al., 2008; Dehne, 2006). In a nutshell, an efficient CGM-based parallel algorithm should reduce the number of communication rounds (which should ideally be constant) and the overall computation time.

1.6 - Dynamic programming

In mathematics, combinatorics studies the configurations of a set of discrete and finite elements to respond to a given problem (Lovász, 2007). Combinatorial problems occur in many areas of mathematics, including algebra, probability theory, topology, and geometry. A well-known combinatorics problem is to determine the number of possible configurations of a given type (graphs, points, arrays). Another combinatorics problem consists in assigning, from an objective function f based on the problem semantics, a numerical value $f(x)$ to any configuration x to choose the best configuration that optimizes (maximizes or minimizes) the function f. This kind of problem is called *combinatorial optimization problems*. Many algorithm design paradigms have been developed to address these problems.

In the 1950s, greedy algorithms were used to solve some problems in data compression (Huffman, 1952) and graph theory (Dijkstra, 1959; Prim, 1957). This strategy attempts to progressively build the best global solution by choosing at each step the best local solution. Later on, Land and Doig (1960) proposed the Branch and Bound strategy. It consists of recursively splitting the search space into subsets and exploring each of them to find feasible solutions. This strategy eliminates candidate solutions that will not lead to an optimal solution based on the properties of the problem. Tabu search is a heuristic method introduced by Glover (1989, 1990). It consists in determining in a neighborhood the best solution that optimizes the objective function at each iteration. One of its strengths is to maintain a tabu list that prevents visiting the solutions already explored. As for the divide-and-conquer strategy, it recursively decomposes a problem into two or more subproblems until they are sufficiently small to be solved. Then, it solves

these subproblems and gradually builds the original solution from their results. This technique was invented by Gauss (1866) when he proposed an algorithm to solve the discrete Fourier transform for interpolating the trajectories of the asteroids Pallas and Juno (Heideman et al., 1984). Several decades later, many problems have been solved using this technique (such as binary search for search problems or quicksort for sorting problems). Nevertheless, this technique has a drawback when used to solve some problems. Indeed, during the problem-solving process, some subproblems are computed as many times as they are encountered. For this reason, Bellman (1957) proposed a strategy that he called *dynamic programming* to solve this kind of problem.

This section describes the dynamic-programming technique. Section 1.6.1 first recalls the divide-and-conquer technique and presents its main drawback based on a counting problem. Then, Section 1.6.2 shows how the dynamic programming overcomes this drawback by defining a solution. Finally, the principles of dynamic programming are outlined in Section 1.6.3.

1.6.1 - Recalling the divide-and-conquer technique

A divide-and-conquer algorithm breaks down a problem into several smaller subproblems that are similar to the original problem, solve them recursively, and then builds the solution of the original problem by combining the solutions of the subproblems (Cormen et al., 2009). There are three main steps in the problem-solving process that are repeated over and over again until the original problem is solved:

Divide the problem into subproblems until the base case of the recursive condition is reached. From then on, the problem is no longer divided.

Conquer the subproblems by solving them.

Combine the solved subproblems together to solve the original problem.

Let us apply this technique to determine the nth Catalan number. In combinatorics, the catalan numbers are a sequence of positive integers that appears in many counting problems (Stanley, 2015), such as :

- counting the number of possible binary search trees with n keys;
- counting the number of ways to parenthesize n objects;
- counting the number of ways to connect the $2n$ points on a circle to form n disjoint chords;

Algorithm 1 Divide-and-conquer algorithm to compute the nth Catalan number

```
1: function CATALAN(n)
2:     if n = 0 or n = 1 then
3:         return 1;
4:     else
5:         c ← 0;
6:         for k = 0 to n − 1 do
7:             c ← c + CATALAN(k) × CATALAN(n − k − 1);
8:         return c;
```

- counting the number of ways to triangulate a convex polygon with n sides;
- counting the number of non-crossing partitions of a set of n elements, or;
- counting the number of semiorders on n unlabeled items.

These problems satisfy the recurrence equation (1.5). It is straightforward to deduce Algorithm 1 from this equation.

$$C_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1, \\ \sum_{k=0}^{n-1} C_k C_{n-k-1} & \text{if } n > 1. \end{cases} \tag{1.5}$$

Figure 8 shows the recursion tree produced by the call `Catalan`(4) to compute C_4, the 4th Catalan number :

1 - C_4 is first divided into two subproblems (C_0 and C_3);

2 - C_0 can be conquered easily but not C_3;

2.1 - C_3 is divided into C_0 and C_2;

2.2 - Since C_2 can't be conquered;

2.2.1 - C_2 is divided into C_0 and C_1;

2.2.2 - Since C_1 can be conquered, C_2 is computed by adding up C_0C_1 and C_1C_0, the combinations of C_0 and C_1;

2.3 - C_0 and C_2 is combined to compute C_0C_2;

2.4 - C_3 is reduced to C_1, which is conquered and combined to obtain C_1C_1;

2.5 - C_3 is divided into C_2 and C_0;

2.6 - Since C_2 can't be conquered;

2.6.1 - C_2 is divided into C_0 and C_1;

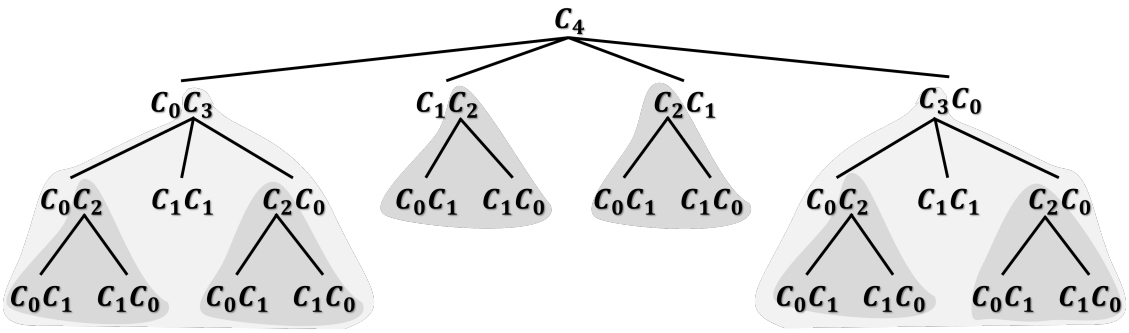


Figure 8 – *Recursion tree while computing the 4th Catalan number using the divide-and-conquer technique*

2.6.2 - Since C_1 can be conquered, C_2 is computed by adding up C_0C_1 and C_1C_0, the combinations of C_0 and C_1;

2.7 - C_2 and C_0 is combined to compute C_2C_0;

2.8 - C_3 is finally computed by adding up C_0C_2, C_1C_1, and C_2C_0;

3 - C_0 and C_3 is combined to obtain C_0C_3.

It is easy to see that steps 2.2 and 2.6 are similar. Indeed, since the result of C_2 is not saved after computing in step 2.2, it must be computed again in step 2.6. In Figure 8, the shaded subtrees represent the subproblems that must be performed more than once when computing C_4. C_3 is computed two times and C_2 is computed six times. This drawback leads to a poor time complexity because the time to compute C_n by this recursive function is exponential in n (Cormen et al., 2009).

1.6.2 - Building a dynamic-programming solution

Dynamic programming (DP) is an extension of the divide-and-conquer technique. Unlike a divide-and-conquer algorithm, a DP algorithm stores the results of subproblems when they are first evaluated. Their results are looked up when these subproblems are encountered the next time during the problem solving process. This allows bounding the number of subproblems to be evaluated and the number of operations to be performed to compute a given subproblem. Thus, it is simpler to determine the time and space complexity of a DP algorithm compared to a divide-and-conquer algorithm, which is usually in polynomial time (Cormen et al., 2009).

There are two approaches to obtaining the solution of the original problem from the results of its subproblems: the top-down approach and the bottom-up approach.

Algorithm 2 Dynamic-programming algorithm to compute the nth Catalan number using the top-down approach

```
1:  function TOPDOWN-CATALAN(n)
2:      let C[0..n] be a table;
3:      for k = 0 to n do
4:          C[k] ← −∞;
5:      return LOOKUP(C, n);

6:  function LOOKUP(C, k)
7:      if C[k] > −∞ then
8:          return C[k];
9:      else
10:         if k = 0 or k = 1 then
11:             C[k] ← 1;
12:         else
13:             C[k] ← 0;
14:             for i = 0 to k − 1 do
15:                 C[k] ← C[k] + LOOKUP(C, i) × LOOKUP(C, k − i − 1);
16:         return C[k];
```

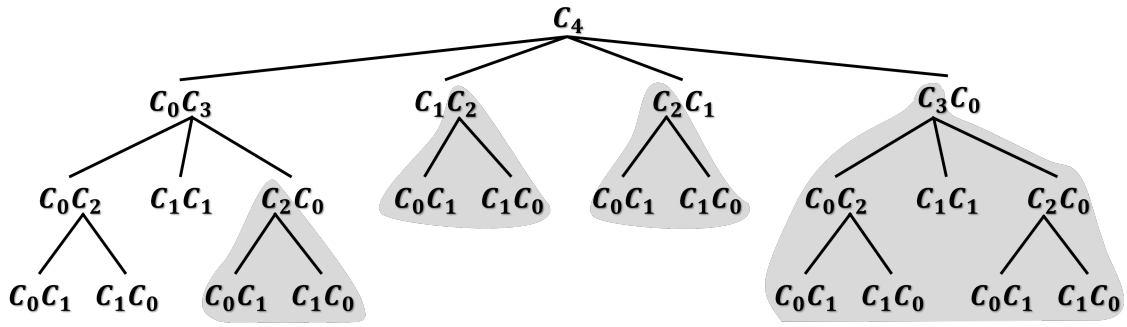
Top-down approach

This approach is similar to the divide-and-conquer method in that it decomposes the original problem until it becomes straightforward to solve. However, it stores the results of subproblems in a table called *dynamic-programming table*. Algorithm 2 computes the nth Catalan number using this approach.

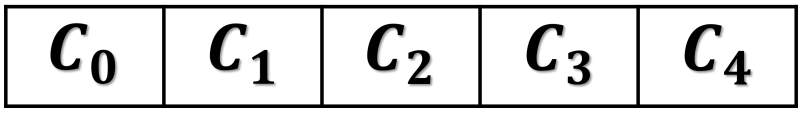
Figure 9a depicts the recursion tree when computing the 4th Catalan number. The shaded subtrees represent the subproblems that are avoided when computing C_4 since their results are looked up in the DP table shown in Figure 9b. Indeed, this table is initialized with a value specifying that the results of subproblems are not yet stored (see lines 3-4 in Algorithm 2). Thereafter, the `Lookup` function is called to retrieve the kth Catalan number from this table. If the result of C_k is present in the DP table then it is returned (see lines 7-8 in Algorithm 2). Otherwise, C_k is recursively computed by calling `Lookup` on its subproblems, stored in the kth entry of the DP table, and returned (see lines 10-16 in Algorithm 2). Compared to Algorithm 1, Algorithm 2 runs in $O\left(n^2\right)$ time since it has $\Theta(n)$ entries in the DP table and each of them makes at most $O(n)$ recursive calls.

Bottom-up approach

This approach progressively solves the smallest subproblems to build the solution of the original problem. It is based on the choice of the right order for solving



(a) *Recursion tree*



(b) *Dynamic-programming table*

Figure 9 – *Recursion tree and the dynamic-programming table while computing the 4th Catalan number using the top-down approach*

Algorithm 3 Dynamic-programming algorithm to compute the nth Catalan number using the bottom-up approach

```
1: function BOTTOMUP-CATALAN(n)
2:     let C[0..n] be a table;
3:     C[0] ← 1;
4:     C[1] ← 1;
5:     for k = 2 to n do
6:         C[k] ← 0;
7:         for i = 0 to k − 1 do
8:             C[k] ← C[k] + C[i] × C[k − i − 1];
9:     return C[n];
```

the subproblems. This order must guarantee that each subproblem is treated only once, and that it is only processed after all the subproblems on which it depends have been computed. For this purpose, the problem is divided into steps. Each step is composed of several subproblems and has a solving strategy. The solution of each of the subproblems of a given step depends only on those of the subproblems belonging to the previous steps, which are stored in the DP table. The subproblems are solved from the first to the last step. The last step solves usually the original problem. Algorithm 3 illustrates the computation of the 4th Catalan number using the bottom-up approach. Since there are $\Theta(n)$ steps and each of them takes at most $O(n)$ time to be solved, this algorithm runs in $O\left(n^2\right)$ time and $O(n)$ space.

Dependencies between the subproblems can be represented by a multi-level, directed and acyclic graph or multi-level DAG (Directed Acyclic Graph) called

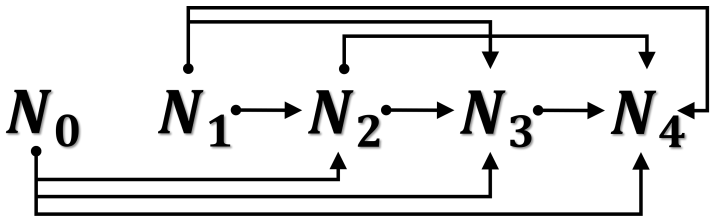


Figure 10 – *Dependencies between subproblems while computing the 4th Catalan number using the bottom-up approach*

task graph or *dependency graph*. Each node of the DAG represents a subproblem. A level (or a step) d is a set of nodes corresponding to the set of subproblems that must be evaluated at step d. A node N_i is necessary for evaluating the node N_j if there is an arc going from N_i to N_j. A node without outgoing (respectively incoming) arcs corresponds to the original (respectively initial) problem. Figure 10 shows the task graph depicting the dependencies between subproblems while computing the 4th Catalan number. The node N_i represents the subproblem C_i and a single node is evaluated at each step. Also, the resolution of the subproblem C_3 depends to already computed values of C_2, C_1, and C_0.

1.6.3 - Principles of dynamic programming

We have applied the dynamic-programming technique to determine the nth Catalan number in Section 1.6.2. In this section, two properties required to apply the DP technique to optimization problems are studied : the optimal substructure and the overlapping subproblems.

Optimal substructure

The optimal substructure property, also called *principle of optimality* (Lew and Mauch, 2007), is the first criterion that a dynamic-programming problem must satisfy. This principle is stated in (Bellman, 1957, pp. 83) as follows :

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

More concretely, this principle states that optimal policies have optimal subpolicies (Lew and Mauch, 2007). This means that each substructure must be optimal within an optimal structure. Hence, whatever the initial state of a problem, the decisions made in the next states must be optimal and consistent with the decisions made in the previous states. Therefore, the optimal solution of a problem is made

up of optimal solutions of its subproblems. A DP formulation seeks to define a problem in this way (Grama et al., 2003).

Some combinatorial problems may have the optimal substructure property, but may use too much memory or time to be efficient. Since the DP technique stores the intermediate results of a problem in a DP table for reuse, computing a solution from them can be time intensive when there are too many intermediate results. For example, a set of n elements has 2^n subsets and there are $n!$ possible permutations of characters of a string of length n. Therefore, the DP technique cannot be efficient to find the best solution for each subpermutation/subset since the DP table would be gigantic. However, it can be efficient only when there are not too many results to compute. For example, there are only $n(n-1)/2$ substrings of a string of length n and only $n(n-1)/2$ possible subtrees of a binary search tree with n keys. In a nutshell, dynamic programming typically produces good results on objects that can be ordered linearly and that cannot be reordered (Kengne, 2014).

Overlapping subproblems

The property of overlapping subproblems is the second criterion that a DP problem must have. This property means that subproblems have to share common subsubproblem; that is, a subsubproblem can belong to several subproblems. In this kind of problem, after evaluating a subsubproblem C generated by a subproblem A, it still needs to be evaluated if it is also generated by a subproblem B. Indeed, C will be evaluated as many times as it has to be generated by a subproblem. Dynamic programming leverages this property to store in a table the results of subproblems that have already been computed. If this subproblem is encountered once again, then it is looked up in the table instead of being recomputed. The fewer subsubproblems there are in common, the faster the DP algorithm will be. Nevertheless, if subproblems generate new subproblems at each step, then it will be more suitable to drop the DP technique (Cormen et al., 2009).

1.7 - Taxonomy of dynamic-programming formulations

The objective function, which characterizes optimization problems, is formulated by a recursive equation called *functional equation* or *optimization equation* (Grama et al., 2003; Kengne, 2014; Lew and Mauch, 2007). The left-hand side represents the unknown value of the optimal solution of a subproblem, and the right-hand side contains the minimization and/or maximization operations on a set of elements (each written in terms of solutions of subproblems belonging to the previous

steps). Defining this equation depends on the semantics of the problem to be addressed. Examples 1 and 2 denote respectively the DP formulations of the all-pairs shortest path problem by Floyd (1962) and the string-to-string correction problem by Wagner and Fischer (1974).

Example 1 *Consider a weighted graph G, which consists of a set of nodes N and a set of edges E. An edge from node i to node j in E has a weight $c_{i,j}$. The cost of a path is the sum of the edges' weights in this path. The algorithm of Floyd (1962) determines the cost $s^k_{i,j}$ of the shortest path between each pair of nodes (i, j) in all intermediate nodes $\{1,2,\ldots,k\}$ through Equation (1.6) defined as follows :*

$$s^k_{i,j} = \begin{cases} c_{i,j} & \text{if } k = 0, \\ \min\left\{s^{k-1}_{i,j}, \left(s^{k-1}_{i,k} + s^{k-1}_{k,j}\right)\right\} & \text{if } 1 \leq k \leq |N| - 1. \end{cases} \tag{1.6}$$

Example 2 *Given two strings $a = a_1 a_2 \ldots a_m$ and $b = b_1 b_2 \ldots b_n$ on an alphabet Σ, the string-to-string correction problem aims to find the distance between a and b as measured by the minimum number of editing operations (insertions, deletions, or substitutions) required to transform a into b. The edit distance between a and b is given by $d_{m,n}$, and defined by Equation (1.7) :*

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & \text{if } a_i = b_j, \\ \min \begin{cases} d_{i-1,j} + deletion(a_i), \\ d_{i,j-1} + insertion(b_j), \\ d_{i-1,j-1} + substitution(a_i, b_j) \end{cases} & \text{if } a_i \neq b_j. \end{cases} \tag{1.7}$$

A DP formulation is said to be *monadic* if the functional equation has only one recurrence term, and it is said to be *polyadic* otherwise (Wah and Li, 1988). The DP formulation in Equation (1.7) is monadic since the functional equation involves one recursive term only. Indeed, the evaluation of the subproblem $d_{i,j}$ depends on only one recursive term (subproblem) at a time : either $d_{i-1,j}$, or $d_{i,j-1}$, or $d_{i-1,j-1}$. In contrast, The DP formulation in Equation (1.6) is polyadic since the evaluation of the subproblem $s^k_{i,j}$ involves two subproblems : $s^{k-1}_{i,k}$ and $s^{k-1}_{k,j}$.

A DP formulation can be also classified according to the nature of the dependency of subproblems. If the solution of a subproblem of a given step depends exclusively on the solutions of subproblems of the immediately preceding step, then the DP formulation is said to be *serial*; otherwise, it is said to be *nonserial* (Wah and Li, 1988). The DP formulation in Equation (1.6) is serial since the evaluation of the subproblem $s^k_{i,j}$ belonging to step k required only the solutions of

subproblems $s_{i,j}^{k-1}$, $s_{i,k}^{k-1}$, and $s_{k,j}^{k-1}$, belonging the previous step $k-1$. In contrast, the DP formulation in Equation (1.7) is nonserial since the evaluation of the subproblem $d_{i,j}$ belonging to step $(j-i-1)$ required the solution of the subproblem $d_{i-1,j-1}$ belonging to step $(j-i-3)$, and the solutions of subproblems $d_{i-1,j}$ and $d_{i,j-1}$ belonging step $(j-i-2)$.

Wah and Li (1988) classified DP formulations into four categories based on the above classification criteria : *serial monadic*, *serial polyadic*, *non-serial monadic*, and *non-serial polyadic*. So, the DP formulation of Floyd (1962) for the all-pairs shortest path problem is part of serial polyadic DP formulations, and the DP formulation of Wagner and Fischer (1974) for the string-to-string correction problem is part of non-serial monadic DP formulations. In this thesis, we are interested in non-serial polyadic dynamic-programming problems, especially in a class of problems whose general DP formulation is described in Section 1.8.

1.8 - General dynamic-programming formulation of the studied problems

Definition 1 *A finite semigroupoid $(S,R,\bullet)$ is a nonempty finite set S of elements, a binary relation $R \subseteq S \times S$, and an associative binary operator $\bullet$ satisfying the following conditions (Marvins, 1978):*

1 - if $(a,b) \in R$, then $a \bullet b \in S$;

2 - $(a \bullet b, c) \in R$ iff $(a, b \bullet c) \in R$ and $(a \bullet b) \bullet c = a \bullet (b \bullet c)$;

3 - if $(a,b) \in R$ and $(b,c) \in R$, then $(a \bullet b, c) \in R$.

Definition 2 *An associative product is any product of the form $a_1 \bullet a_2 \bullet \cdots \bullet a_n$ such that $(a_i, a_{i+1}) \in R$, $1 \leq i \leq n$. A linear product is a product of the form $((\cdots(a_1 \bullet a_2) \bullet \cdots) \bullet a_n)$ or $(a_1 \bullet (\cdots \bullet (a_{n-1} \bullet a_n)) \cdots)$.*

Definition 3 *A weighted semigroupoid $(S,R,\bullet,pc)$ is a semigroupoid $(S,R,\bullet)$ with a non-negative product cost function pc. If $(a_i,a_j) \in R$ then $pc(a_i,a_j)$ is the cost of evaluating $a_i \bullet a_j$. The optimal cost of evaluating an association product $a_i \bullet a_{i+1} \bullet \cdots \bullet a_j$ is denoted by $Cost[i,j]$ and defined by:*

$$Cost[i,j] = \begin{cases} Init(i) & \text{if } 1 \leq i = j \leq n, \\ Opt_{i \leq k < j}\{Cost[i,k] + Cost[k+1,j] + F(i,k,j)\} & \text{if } 1 \leq i < j \leq n, \end{cases} \tag{1.8}$$

where n is the problem size and $F(i,k,j) = pc(a_i \bullet \cdots \bullet a_k, a_{k+1} \bullet \cdots \bullet a_j)$.

Equation (1.8) shows that the problems studied in this thesis feature the properties of a DP problem :

- Optimal substructure property : the optimal costs of the subproblems (k,l) are required for computing the optimal cost of a subproblem (i,j), where $i \leq k < l \leq j$.
- Overlapping subproblems property : consider two subproblems (i,j) and $(i+1,j)$. The evaluation of these subproblems depends on the common values of the evaluation of subproblems (k,j), where $i+1 < k \leq j$. It is the same if two subproblems (i,j) and $(i,j+1)$ are considered.

In Equation (1.8), $Cost[i,j]$ corresponds to the value of the optimal solution of the subproblem (i,j). This value is obtained according to the optimum function Opt (for a maximization problem, $Opt = \max$; otherwise $Opt = \min$) among the $(j-i)$ possible combinations of the subsubproblems on which the subproblem (i,j) depends. The value of a combination of two subproblems (i,k) and $(k+1,j)$, where $i \leq k < j$, is computed by adding the values of their optimal solutions and the cost corresponding to the combination of these subproblems given by the function $F(i,k,j)$ called *union function*. The basic subproblems are initialized by the function $Init(i)$.

Several problems can be formulated from Equation (1.8) such as the context-free grammar parsing problem (Kasami, 1965; Younger, 1967) and the Nussinov RNA folding problem (Nussinov and Jacobson, 1980). In this thesis, we are interested in the minimum cost parenthesizing problem (MPP), the matrix chain ordering problem (MCOP), the triangulation of a convex polygon (TCP) problem, and the optimal binary search tree (OBST) problem. The common feature of these problems is the minimization of the optimization equation (this means that $Opt = \min$ in Equation (1.8)). However, they differ in the definition of the union function F and initialization function $Init$ because the definition of these functions depends on the problem to solve.

A solution based on the exhaustive search of all possible combinations would be poor because, for a problem of size n, the number of combinations is exponential in n (see Section 1.6.1). To solve the MCOP, Godbole (1973) proposed the first polynomial-time sequential algorithm running in $O(n^3)$ time and $O(n^2)$ space. It became the standard algorithm for solving all problems that can be formulated by Equation (1.8) because the structure and the complexity of this algorithm are

Algorithm 4 Generic sequential algorithm to solve the studied problems

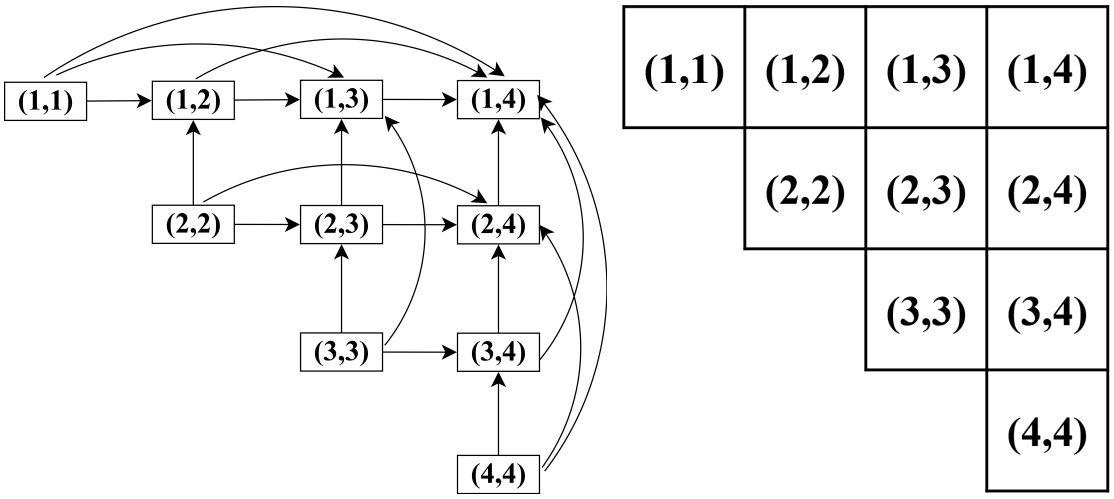
```
1: for i = 1 to n do
2:     Cost[i,i] ← Init(i);
3: for d = 2 to n do
4:     for i = 1 to n − d + 1 do
5:         j ← n − d + 1;
6:         Cost[i, j] ← ∞;
7:         for k = i to j − 1 do
8:             c ← Cost[i,k] + Cost[k + 1, j] + F(i,k, j);
9:             if c < Cost[i, j] then
10:                Cost[i, j] ← c;
11:                Track[i, j] ← k;
```

independent of the functions F and $Init$. This algorithm is often called *generic sequential algorithm* in the literature (Kengne, 2014). Algorithm 4 draws the big picture.

Computing $Cost[1,n]$ involves the solution of all subproblems (i, j), such that $1 \leq i \leq j \leq n$. Dependencies between these subproblems can be organized like a DAG, as shown in Figure 11a for a problem of size $n = 4$. This figure reveals, for example, that the value of $Cost[1,4]$ is computed from the optimal solutions of the pairs of subproblems $((1,1),(2,4))$, $((1,2),(3,4))$, and $((1,3),(4,4))$. Indeed, Algorithm 4 uses a bottom-up approach to compute the optimal solutions of subproblems. The value of the optimal solution of each solved subproblem is stored in the DP table depicted in Figure 11b. The index k that led to this value among the $(j-i)$ possible solutions for a subproblem (i, j) is also saved in a table called *tracking table*. This table, named *Track* in Algorithm 4, is similar to the DP table and is used for the construction of the optimal solution.

1.9 - Summary

This chapter was dedicated to the definition of the basic concepts of parallel computing and dynamic programming. In the first part, we started by giving the principle of high-performance computing, which is the *raison d'être* of parallel computing. Then, after presenting the classifications of parallel computer architectures according to several criteria, such as the number of instruction streams and data streams, the memory of parallel computers, the network topology, and the granularity, we studied different techniques to parallelize a sequential algorithm and showed how to measure the performance of a parallel algorithm. Finally, we stud-



(a) *Task graph* **(b)** *Dynamic-programming table*

Figure 11 – *Task graph and dynamic-programming table used to compute* $Cost[1,4]$

ied parallel computing models. Different problems of too abstract models like the PRAM model and too realistic models like the systolic model and the hypercube model have been highlighted. The BSP model solved the problems of previous models and the CGM model simplified the BSP model.

In the second part, we first recalled the divide-and-conquer technique and showed its main drawback before presenting the top-down approach and the bottom-up approach of dynamic programming, which overcome this drawback. We then gave the principles of dynamic programming. After classifying the DP formulations, we presented the formulation of the class of non-serial polyadic dynamic-programming problems that interest us.

Chapter 2 will study in detail the parallelization of the sequential algorithms that solve each of these problems.

CHAPTER 2

Parallelization of the Studied Problems : State of the Art

CONTENTS

2.1 - Introduction

This chapter reviews the state of the art on parallelization of sequential algorithms for the minimum cost parenthesizing problem, the matrix chain ordering problem, the triangulation of a convex polygon problem, and the optimal binary search tree problem. For each of them, the following points will be presented and studied successively: the description of the problem and its dynamic-programming (DP) formulation, the best sequential algorithm that solves this problem, the parallelization constraints and the literature review on the parallelization of this sequential algorithm on parallel computing models, and finally, the best CGM-based parallel solutions. For the special case of the matrix chain ordering problem, we will present our fast sequential algorithm and experimental results obtained.

2.2 - Minimum cost parenthesizing problem

2.2.1 - Overview

A parenthesizing process is any process that inserts opening and closing parentheses between a chain of symbols (characters, numbers, matrices, or objects) in

order to define the best sequential processing order according to an intended goal (Kengne, 2014). The minimum cost parenthesizing problem (MPP) of a chain of symbols consists in finding the parenthesizing that will minimize the cost of the computations involved on this chain. Depending on the kind of entities in the chain to parenthesize and the treatment to perform (see the general formulation in Section 1.8), this problem appears in the literature under several variants. A well-known variant is the parenthesizing problem of the lexical ordering of the computation blocks for embedded DSP (Digital Signal Processor) applications (Bhattacharyya and Murthy, 1995). The goal is to find the parenthesizing that minimizes the memory requirement on the DSP. The most popular variant is the matrix chain ordering problem (MCOP), which consists in finding the parenthesizing that minimizes the cost of the product of a chain of matrices (Cormen et al., 2009; Kengne, 2014).

First, consider the cost of multiplying two matrices A and B with respective dimensions $(a_0 \times a_1)$ and $(b_0 \times b_1)$. They can be multiplied only if $a_1 = b_0$. This operation requires $a_0 \times c \times b_1$ scalar multiplications, where $c = a_1 = b_0$, and produces in output a matrix C with dimensions $(a_0 \times b_1)$. The way in which a chain of matrices is bracketed can have a significant impact on the evaluation of the product cost. To illustrate this, consider four matrices M_1, M_2, M_3, and M_4 with respective dimensions (5×10), (10×3), (3×20), and (20×6). Evaluating their product can be done in different ways:

1 - $(M_1 \times (M_2 \times M_3)) \times M_4$, which requires $10 \times 3 \times 20 + 5 \times 10 \times 20 + 5 \times 20 \times 6 = 2200$ scalar multiplications;

2 - $M_1 \times (M_2 \times (M_3 \times M_4))$, which requires $3 \times 20 \times 6 + 10 \times 3 \times 6 + 5 \times 10 \times 6 = 2460$ scalar multiplications;

3 - $(M_1 \times M_2) \times (M_3 \times M_4)$, which requires $5 \times 10 \times 3 + 3 \times 20 \times 6 + 5 \times 3 \times 6 = 600$ scalar multiplications.

Thus, the product cost depends on the chosen order although all bracketings yield the same output matrix M with dimensions (5×6)[1]. Therefore, the MCOP consists in finding the optimal order having the lowest cost for multiplying matrices. The input is a list of $(n+1)$ natural numbers $d_0, d_1, \ldots, d_n$, representing the dimensions of the matrices (a matrix M_i has dimensions $(d_{i-1} \times d_i)$). The output is the minimum cost to compute the product of the n matrices.

Note that the MCOP does not multiply the matrices. It simply looks for an order of multiplication that minimizes the cost. The time spent to determine this optimal

1. The matrix multiplication is associative.

order is commonly offset by the time saved when performing matrix multiplications using this optimal order; for example, performing 600 scalar multiplications will theoretically take four less time than 2460 (Cormen et al., 2009). Traditionally, the matrix multiplication was introduced in linear algebra to facilitate and clarify computations (Marvins, 1978; Schreier and Sperner, 2011). This strong relationship between matrix multiplication and linear algebra remains fundamental in mathematics, as well as in physics, engineering, and computer science, more precisely in scientific computing (Lee et al., 2003; Lin, 1994; Yau and Lu, 1993).

Back to the MCOP, the minimum cost for evaluating $M_{i,j} = M_i \times M_{i+1} \times \cdots \times M_j$ is denoted by $Cost[i,j]$ and defined by:

$$Cost[i,j] = \begin{cases} 0 & \text{if } 1 \leq i = j \leq n, \\ \min\limits_{i \leq k < j} \{Cost[i,k] + Cost[k+1,j] + d_{i-1} \times d_k \times d_j\} & \text{if } 1 \leq i < j \leq n. \end{cases} \tag{2.1}$$

Solving $M = M_1 \times M_2 \times \cdots \times M_i \times \cdots \times M_n$ reduces to computing $Cost[1,n]$. Equation (2.1) is equivalent to Equation (1.8) as $Opt = \min$, $Init(i) = 0$, and $F(i,k,j) = d_{i-1} \times d_k \times d_j$, where $1 \leq i \leq k \leq j \leq n$.

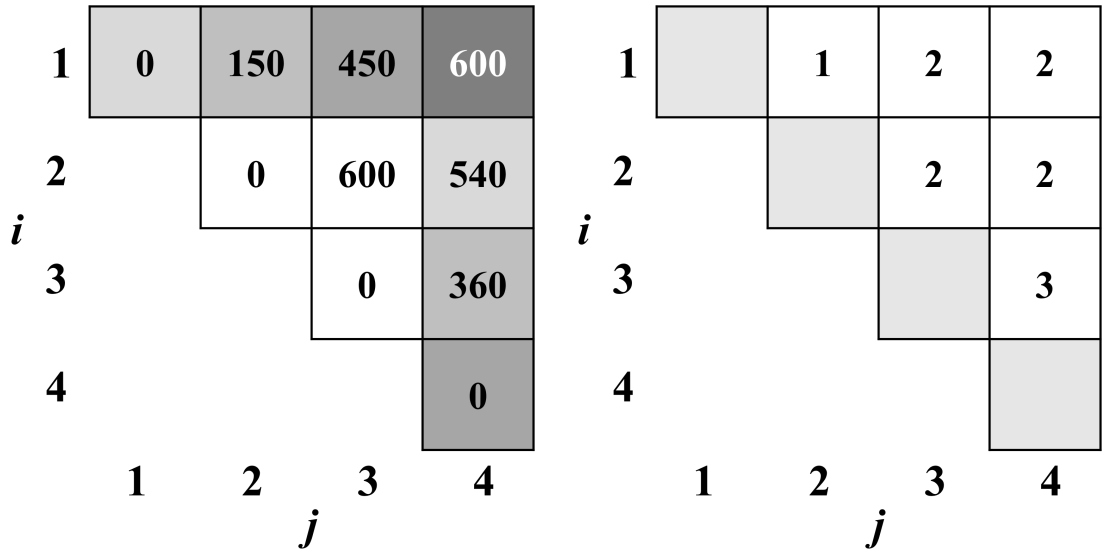
2.2.2 - Sequential algorithm of Godbole (1973)

Algorithm 5 gives an overview of the sequential algorithm of Godbole (1973) to solve the MCOP. Since the optimal product cost of $M_{i,j}$ never depends on a product of length greater than $(j-i)$, the subproducts can be organized and computed in n levels (steps or diagonals) according to their length. Thus, products of length d, which correspond to $M_{i,i+d-1}$ such that $1 \leq i \leq i+d-1$, are evaluated at diagonal d in Algorithm 5. The solution of Godbole (1973) aims at filling, diagonal by diagonal, in cells of the DP table (named *Cost* in Algorithm 5) and the tracking table (named *Track* in Algorithm 5) respectively the value of the optimal product cost of $M_{i,j}$ (see line 10 in Algorithm 5) and the value of the index k corresponding to the pair of subproducts $(M_{i,k}, M_{k+1,j})$, which minimizes $Cost[i,j]$ (see line 11 in Algorithm 5). The first diagonal of the DP table is initialized to 0 since there is no product of length 1 (see lines 1-2 in Algorithm 5).

Figures 12a and 12b illustrate, respectively, the DP table and the tracking table that have been filled while computing the product of four matrices with respective dimensions (5×10), (10×3), (3×20), and (20×6). Each table is a square matrix of order $n = 4$. However, only the part of the table strictly above the main diagonal (the first diagonal) is used. It is obvious that one can use a single table in which the part of the table above the main diagonal will be used to store the values of

Algorithm 5 Sequential algorithm of Godbole (1973) to solve the MCOP

```
1:  for i = 1 to n do
2:      Cost[i,i] ← 0;
3:  for d = 2 to n do
4:      for i = 1 to n − d + 1 do
5:          j ← n − d + 1;
6:          Cost[i, j] ← ∞;
7:          for k = i to j − 1 do
8:              c ← Cost[i,k] + Cost[k + 1, j] + d_{i−1} × d_k × d_j;
9:              if c < Cost[i, j] then
10:                 Cost[i, j] ← c;
11:                 Track[i, j] ← k;
```



(a) *Dynamic-programming table* **(b)** *Tracking table*

Figure 12 – *Dynamic-programming and tracking tables filled while computing the product of four matrices with respective dimensions* (5×10), (10×3), (3×20), *and* (20×6)

$Cost[i, j]$, and the part of the table below the main diagonal will be used to store the values of $Track[j, i]$. The shaded entries in Figure 12a depict the computation of $Cost[1,4]$. The pairs that have the same shading are taken together in line 8 of Algorithm 5 when computing

$$Cost[1,4] = \min \begin{cases} Cost[1,1] + Cost[2,4] + d_0 \times d_1 \times d_4 = 0 + 540 + 300 = 840, \\ Cost[1,2] + Cost[3,4] + d_0 \times d_2 \times d_4 = 150 + 360 + 90 = 600, \\ Cost[1,3] + Cost[4,4] + d_0 \times d_3 \times d_4 = 450 + 0 + 600 = 1050 \end{cases}$$
$$= 600.$$

To solve the MCOP, the sequential algorithm of Godbole (1973) requires $O\left(n^3\right)$ time. This is deducible since the loops are nested at three deep, and each of them takes at most $(n-1)$ values. It requires $O\left(n^2\right)$ memory space to store the DP table and the tracking table.

Parallelization constraint of the sequential algorithm of Godbole (1973)

The main parallelization constraint of this algorithm comes from the irregular computational load for the evaluation of subproducts of a diagonal d since the evaluation of subproducts of length d is smaller than subproducts of length $(d+1)$ (it has more operations to evaluate the subproblems of length $(d+1)$ than those of length d). Thus, in the parallelization process, this irregular computational load between the diagonals of the sequential algorithm of Godbole (1973) must be considered. It can cause the load imbalance between processors and lead to poor efficiency of the parallel algorithm. Some research has investigated the parallelization of this sequential solution on different parallel computing models in the literature.

Literature review on the parallelization of the sequential algorithm of Godbole (1973)

On the PRAM model, Valiant et al. (1983) proposed a general method for parallelizing non-serial polyadic dynamic-programming problems. It requires $\Theta\left(\lg^2 n\right)$ computation time on $O\left(n^9\right)$ processors. Rytter (1988) reduced the number of processors down to $O\left(n^6/\lg n\right)$ by using the specific features of this class of problems. Later on, Huang et al. (1994) improved this solution by reducing the number of processors to $O\left(n^6/\log^5 n\right)$. On CREW-PRAM machines, Bradford (1994) and Czumaj (1993) designed solutions running in $O\left(\log^3 n\right)$ time using respectively $O\left(n^3/\log n\right)$ and $O\left(n^2/\log^3 n\right)$ processors. Tang and Gupta (1995) presented an algorithm requiring $\Theta(n)$ computation time with $\Theta\left(n^2\right)$ processors. Ramanan (1996) is the first to design an algorithm using only $O(n)$ processors with $O\left(\log^3 n\right)$ time. Based on the row minima of totally monotone matrices, Bradford et al. (1998) proposed the first parallel polylogarithmic algorithm. It requires $O\left(n\log^{1.5} n\right)$ time and processors. On CRCW-PRAM machines, Bradford (1994) proposed an $O\left(\lg^4 n\right)$-time algorithm on $O(n/\lg n)$ processors. Later on, Bradford et al. (1998) improved it to $O\left(\lg^{1.5} n\sqrt{\lg\lg n}\right)$ time using $O\left(n\sqrt{\lg n}\right)$ processors. On the systolic model, Guibas et al. (1979) proposed a bidirectional systolic algorithm on $O\left(n^2\right)$ processors with $O(n)$ running time. Karypis and Kumar (1993) mapped a systolic table onto a mesh-connected array of size n^2. Myoupo (1992, 1993) proposed a technique for mapping the MCOP to a linear systolic array. On

the hypercube model, Ibarra et al. (1991) presented a solution in $O\left(n^2\right)$ time on $O(n)$ processors.

On realistic models of parallel machines, Nishida et al. (2011) presented an efficient parallel implementation of the sequential algorithm of Godbole (1973) on GPU architectures. Ito and Nakano (2013) accelerated this solution. Their idea consists in partitioning the sequential algorithm of Godbole (1973) into many sequential kernel calls, selecting the best values for the size and the number of blocks for each kernel call, and minimizing the memory access overhead. On a NVIDIA GeForce GTX 680 chip, their solution executes in 5.57 seconds for a problem of size 8192. Shyamala et al. (2017) accelerated the computation time through C++ high-performance accelerated massive parallel code. More recently, Diwan and Tembhurne (2019) designed an adaptive generalized mapping method to parallelize non-serial polyadic dynamic-programming problems that utilize GPUs, for efficient mapping of subproblems onto processing threads in each phase. Biswas and Mukherjee (2021) proposed a new memory optimized technique and a versatile technique of utilizing shared memory in blocks of threads to minimize time for accessing dimensions of matrices on GPU architectures. On shared-memory architectures, Tan et al. (2007) proposed a parallel algorithm based on a pipeline to fill the DP table by decomposing the computation operators. Mabrouk (2016) designed solutions based on loop transformations. She showed that the associative expression evaluation technique with the loop interchange transformation provides efficient parallel solutions. Many researchers proposed parallel solutions on the CGM model on distributed-memory architectures. For designing these solutions, the standard methodology consists of subdividing the dependency graph into subgraphs of same size, then distributing these subgraphs fairly among processors, and finally computing them in a suitable evaluation order.

Based on the sequential algorithm of Godbole (1973), Cáceres et al. (2010) proposed a CGM-based parallel solution running in $O\left(n^3/p\right)$ execution time with $O\left(p\right)$ communication rounds, where n is the input data size and p the number of processors. Higa and Stefanes (2012) used several concepts from the PRAM-based parallel algorithm of Bradford (1994) and designed an $O\left(n^3/p^3\right)$ execution time with $O\left(1\right)$ communication round. They replaced the traditional DP table used in the sequential algorithm of Godbole (1973) with a weighted DAG. The processors used in the local computation the algorithm of Dijkstra (1959) and the algorithm of Floyd (1962). Kechid and Myoupo (2009) proposed a CGM-based parallel solution, which runs in $O\left(n^3/p\right)$ execution time with $O(p)$ communication rounds. Likewise, the CGM-based parallel solution proposed by Kengne and My-

oupo (2012) runs in $O\left(n^3/p\right)$ execution time with $\lceil\sqrt{2p}\rceil$ communication rounds. The common point between these last two solutions is that they use the graph model proposed by Bradford (1994). Before detailing the latter two solutions in Sections 2.2.4 and 2.2.5 respectively, the graph model proposed by Bradford (1994), which enables to transform the MCOP as a shortest path problem, is presented in Section 2.2.3.

2.2.3 - Dynamic graph model of Bradford (1994)

Dynamic-programming problems are often solved through the shortest path problems on weighted DAGs. Indeed, Bradford (1994) proposed a graph model called *dynamic graph* for the problem that can be formulated by Equation (1.8). For a problem of size n, this graph is denoted by D_n and defined as follows :

Definition 4 *A dynamic graph $D_n = (V, E \cup E')$ is defined as a set of vertices,*

$$V = \{(i,j) : 1 \leq i \leq j \leq n\} \cup \{(0,0)\}$$

a set of unit edges,

$$E = \{(i,j) \rightarrow (i,j+1) : 1 \leq i \leq j < n\} \cup \{(i,j) \uparrow (i-1,j) : 1 < i \leq j \leq n\} \cup \{(0,0) \nearrow (i,i) : 1 \leq i \leq n\}$$

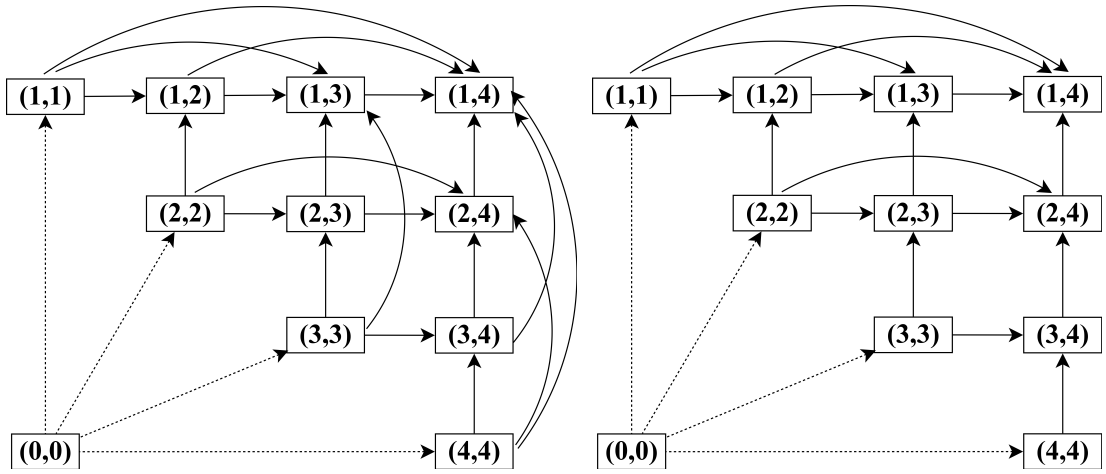
a set of jumpers,

$$E' = \{(i,j) \Rightarrow (i,t) : 1 \leq i < j < t \leq n\} \cup \{(s,t) \Uparrow (i,t) : 1 \leq i < s < t \leq n\}$$

a weight function W such that

$$\begin{array}{lll}
W((i,j) \rightarrow (i,j+1)) & = d_{i-1} \times d_j \times d_{j+1} & 1 \leq i \leq j < n \\
W((i,j) \uparrow (i-1,j)) & = d_{i-2} \times d_{i-1} \times d_j & 1 < i \leq j \leq n \\
W((0,0) \nearrow (i,i)) & = 0 & 1 \leq i \leq n \\
W((i,k) \Rightarrow (i,j)) & = SP[k+1,j] + d_{i-1} \times d_k \times d_j & 1 \leq i < k < j \leq n \\
W((k+1,j) \Uparrow (i,j)) & = SP[i,k] + d_{i-1} \times d_k \times d_j & 1 \leq i \leq k < j \leq n
\end{array}$$

A square matrix of size n called *shortest path matrix*, and denoted by SP, is used to store in the cell $SP[i,j]$ the shortest path from node $(0,0)$ to (i,j). Bradford (1994) showed that the computation of $Cost[i,j]$ is equivalent to search in D_n the shortest path from node $(0,0)$ to (i,j). Therefore, it is straightforward to prove that the algorithm of Godbole (1973) presented in Section 2.2.2 (Algorithm 5) is equivalent to compute the shortest paths from $(0,0)$ to the other vertices in a dynamic graph D_n, incrementally, diagonal after diagonal, from left to right (Kechid and



(a) *Dynamic graph* D_4 **(b)** *Dynamic graph* D'_4

Figure 13 – *Dynamic graphs* D_4 *and* D'_4 *for a problem of size* $n = 4$

Myoupo, 2009; Kengne and Myoupo, 2012). Indeed, each path from the root to an edge node corresponds to one of the possible parenthesizes in a dynamic graph given that :

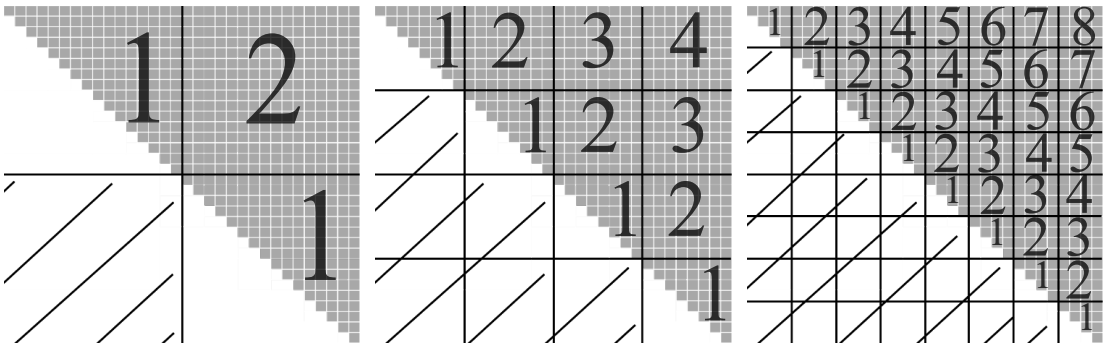
- $(i, j) \rightarrow (i, j+1)$ represents the product $(M_i \times M_{i+1} \times \cdots \times M_j) \times M_{j+1}$;
- $(i, j) \uparrow (i-1, j)$ represents the product $M_{i-1} \times (M_i \times M_{i+1} \times \cdots \times M_j)$;
- $(i,k) \Rightarrow (i, j)$ and $(k+1, j) \Uparrow (i, j)$ represent the product $(M_i \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j)$.

Thus, the shortest path corresponds to the optimal parenthesizing. Figure 13a shows a dynamic graph D_n for $n = 4$. It has the same DAG form representing the dependency between subproblems depicted in Figure 11a, with an additional node $(0,0)$.

Given a problem of size *n* and its corresponding dynamic graph D_n, Theorem 1 can be stated.

Theorem 1 (Duality theorem) *If the shortest path from node* $(0,0)$ *to* (i, j) *needs the edge from node* (i,k) *to* (i, j)*, then there exists a dual shortest path with the same cost needing the edge from node* $(k+1, j)$ *to* (i, j).

This is a fundamental element of CGM-based parallel algorithms proposed by (Kechid and Myoupo, 2009; Kengne and Myoupo, 2012) because it avoids computation redundancy of the shortest path costs in D_n. The input graph of these algorithms is therefore a subgraph of D_n denoted by D'_n, in which the set of edges



(a) $p = 2$ **(b)** $p = 4$ **(c)** $p = 8$

Figure 14 – *Shortest path matrix partitioning strategy proposed by Kechid and Myoupo (2009) for* $n = 32$ *and* $p \in \{2,4,8\}$

from node $(k+1, j)$ to (i, j), such that $1 \leq i < k < j \leq n$, is removed. Figure 13b shows the dynamic graph D'_4.

2.2.4 - CGM-based parallel solution of Kechid and Myoupo (2009)

Dynamic graph partitioning

Kechid and Myoupo (2009) partition the shortest path matrix SP into $p(p+1)/2$ submatrices called *blocks*. A block, denoted by $SM(i, j)$, is a matrix of size $\theta(n,p) \times \theta(n,p)$, where $\theta(n,p) = \lceil n/p \rceil$. Equation (2.2) shows entries of the shortest path matrix SP delimiting a block $SM(i, j)$, such that $1 \leq i \leq j \leq n$.

$$SM(i,j) = \begin{pmatrix} SP[i, j-\theta(n,p)+1] & \cdots & SP[i,j] \\ \vdots & \cdots & \vdots \\ SP[i+\theta(n,p)-1, j-\theta(n,p)+1] & \cdots & SP[i+\theta(n,p)-1, j] \end{pmatrix} \tag{2.2}$$

Figures 14a, 14b, and 14c depict three scenarios of this partitioning for $n = 32$ and $p \in \{2,4,8\}$ (the node $(0,0)$ and unit edges of the dynamic graph D'_{32} have not been schematized for readability reasons). In each of them, SP is subdivided into p rows and p columns of blocks. The number of blocks in row i (respectively in column j) is $p-i+1$ (respectively j). Usually, all blocks are not full when $n \bmod p \neq 0$. Recall that a block is said *full* when the number of rows is equal to the number of columns. In these figures, the number in each block represents the diagonal in which it belongs. The number of blocks in the diagonal d is $p-d+1$. Thus, there are p blocks at the first diagonal and one block at the last. The blocks of the first diagonal are upper triangular matrices of sizes $\theta(n,p) \times \theta(n,p)$.

Blocks' dependency analysis

Kechid and Myoupo (2009) showed that the evaluation of shortest paths for different node blocks of the same diagonal can be carried out in parallel. Indeed, the dependency relationship between blocks, given by Theorem 2, proved that those on the same diagonal are independent.

Lemma 1 (Nodes' dependency) *To find the shortest path cost to a node (i,j) in graph D'_n, it is necessary to know the shortest path cost to nodes $(i,i),\ldots,(i,j-1)$ and $(i+1,j),\ldots,(j,j)$.*

Proof. To compute the shortest path to a vertex (i,j), it is necessary, for each of its incoming edges, to have the value of the shortest path to the starting vertex and the weight of this edge. In the dynamic graph D'_n, the edges that reach a vertex (i,j) come from the set of vertices : $S'_{i,j} = (i,i),\ldots,(i,j-1) \cup (i+1,j)$. The weight of a jump is $W((i,k) \Rightarrow (i,j)) = SP[k+1,j] + d_{i-1} \times d_k \times d_j$. The computation of weights of edges coming from the vertices of $S'_{i,j}$ requires the values of the shortest paths to the vertices $S''_{i,j} = (i+1,j),\ldots,(j,j)$ of the column j. Thus, the computation of $SP[i,j]$ involves the values of the shortest paths to the vertices of the groups $S'_{i,j}$ and $S''_{i,j}$. ■

Lemma 2 (Weights of jumps to a block) *Computing the weights of jumps from nodes of block $SM(i,h)$ to nodes of block $SM(i,j)$, such that $1 \leq i \leq h \leq j \leq n$, only involves the shortest path costs to nodes of block $SM(h-\theta(n,p)+2,j)$.*

Proof. Computing the weights of jumps from a set of successive nodes in the row i, $\{(i,h),(i,h+1),\ldots,(i,l)\}$, to a node (i,m) such that $i \leq h < l < m$, requires only the shortest path costs to nodes $\{(h+1,m),\ldots,(l+1,m)\}$, which is a set of successive nodes of column m. Thus, computing the weight of any row i' of block $SM(i,h)$, i.e. $(i',h-\theta(n,p)+1),(i',h-\theta(n,p)+2),\ldots,(i',h)$ to each node (i',j') of row i' in block $SM(i,j)$, i.e. $(i',j-\theta(n,p)+1),(i',j-\theta(n,p)+2),\ldots,(i',j'),\ldots,(i',j)$, requires only the shortest path costs to the set of nodes: $\{(h-\theta(n,p)+2,j'),(h-\theta(n,p)+3,j'),\ldots,(h+1,j')\}$.

But this set is exactly the set of vertices of column j' in block $SM(h-\theta(n,p)+2,j)$ with $j' \in \{(j-\theta(n,p)+1),(j-\theta(n,p)+2),\ldots,(j-1),j\}$. Thus, computing the weights of jumps of each row in $SM(i,h)$ requires only the shortest path costs of nodes in block $SM(h-\theta(n,p)+2,j)$. ■

Theorem 2 (Blocks' dependency) *The shortest path costs to every node in blocks $SM(i,j-\theta(n,p)),SM(i,j-2\times\theta(n,p)),\ldots,SM(i,j-(u-1)\times\theta(n,p))$, and $SM(i$*

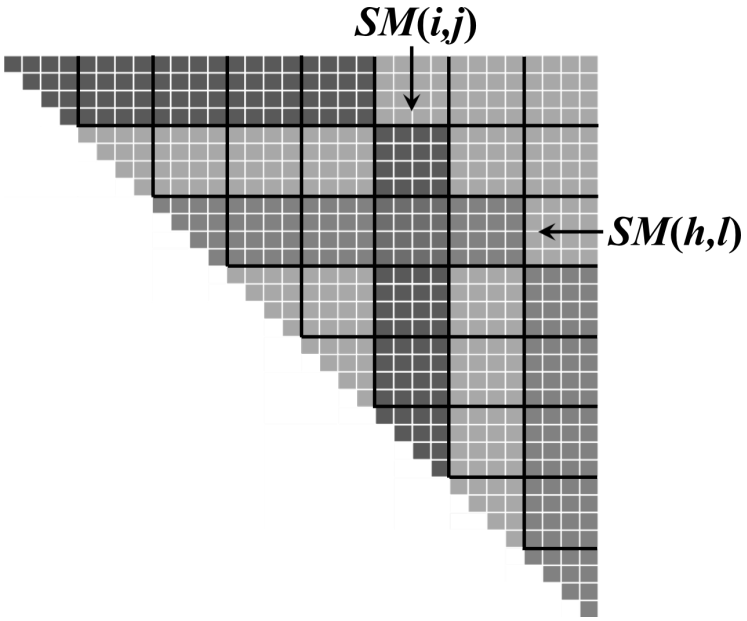


Figure 15 – *Dependencies of two blocks $SM(i,j)$ and $SM(h,l)$ after applying the partitioning strategy of Kechid and Myoupo (2009)*

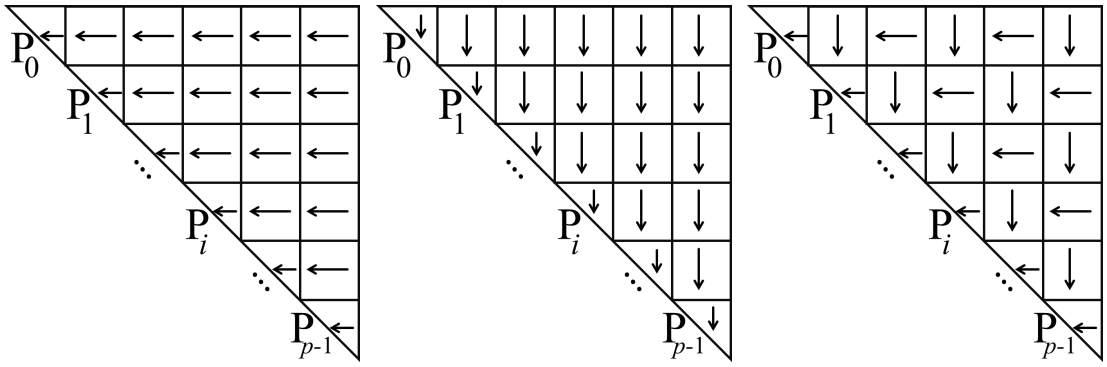
$+\theta(n,p),j)$, $SM(i+2\times\theta(n,p),j),\ldots,SM(i+(u-1)\times\theta(n,p),j)$ *are mandatory to computing the shortest paths to nodes of the block* $SM(i,j)$, *where* $u=\lceil(j-i)/\theta(n,p)\rceil$.

Figure 15 illustrates an example of dependencies of two blocks $SM(i,j)$ and $SM(h,l)$. The most shaded blocks are required to evaluate them.

Mapping blocks onto processors

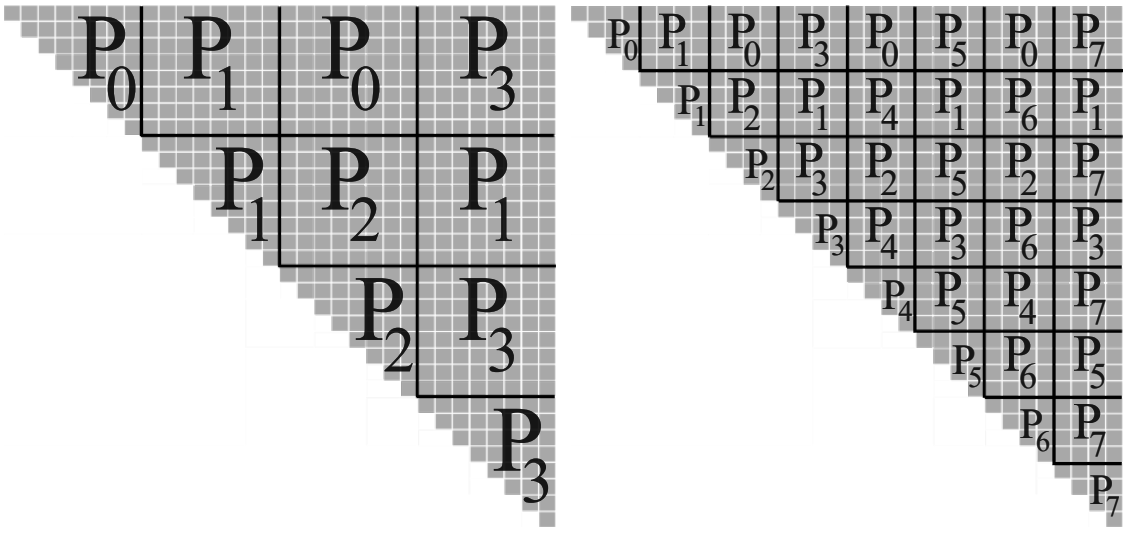
Several distribution schemes have been proposed by Kechid and Myoupo (2009). The first is a straightforward mapping that consists in assigning, horizontally (respectively vertically), the blocks of the line (respectively column) i to processor P_{i-1}. The corresponding communication scheme, depicted in Figure 16a (respectively in Figure 16b), is easily implementable. However, these mappings make processors idle since after each step, one processor will not have assigned block. For example, Figure 16a (respectively Figure 16b) shows that the processor P_{p-1} (respectively P_0) will be idle after the first step, then it will be the turn of the processor P_{p-2} (respectively P_1) after the second step, and so forth. Moreover, these mappings yield an unbalanced load between them since the processor having the highest load will evaluate p blocks and the one having the lowest load will evaluate one block.

The second mapping, called *alternative bidirectional projection mapping*, facilitates the cost prediction and implementation of the algorithm while trying to



(a) *Horizontal mapping* **(b)** *Vertical mapping* **(c)** *Alternative bidirectional projection mapping*

Figure 16 – *Distribution schemes of blocks on processors proposed by Kechid and Myoupo (2009)*



(a) $p = 4$ **(b)** $p = 8$

Figure 17 – *Alternative bidirectional projection mapping on four and eight processors*

balance the loads between processors. Its principle is illustrated in Figure 16c. An arrow from a block points to the processor to which it is assigned. To distribute the $p(p+1)/2$ blocks of the dynamic graph D'_n on p processors, one of the p blocks of the first diagonal is assigned to a processor. The blocks on other diagonals are projected onto the first diagonal, alternatively, once horizontally and once vertically. This approach allows obtaining horizontal and vertical arrows. Figures 17a and 17b give an example of this mapping on four and eight processors.

The communication scheme corresponding to this mapping is simple and easy to implement like the latter two mappings. However, this mapping halves the number of processors to which a block is sent after being computed. Indeed, a processor owns half of the blocks on the row to which it has been assigned its first block of

the first diagonal, moving from left to right. It also owns half the blocks of the blocks on the column by moving from bottom to top. In addition, it balances the loads between processors better than the last two mappings for the reasons stated above.

The alternative bidirectional projection mapping, by contrast, promotes the idleness of processors from step $(\lceil p/2 \rceil + 1)$. Indeed, starting from this step, processor $P_{\lceil p/2 \rceil}$ becomes idle; for example in Figure 17b, processor P_4 after step 4 becomes idle. Subsequently, after each step, two new processors become idle; for example in Figure 17b, processors P_2 and P_3 become idle after step 5. In addition, some processors from step $(\lceil p/2 \rceil + 1)$ have a large computational load while others are idle; for example processors P_0, P_1, and P_7 in Figure 17b, which have two blocks to be performed after step 4.

CGM-based parallel algorithm

Evaluating blocks of the dynamic graph D'_n must be done according to a well-adapted order following dependencies between them. Indeed, according to Theorem 2, the shortest path costs to nodes of a given block belonging to the diagonal d cannot be performed before those contained in each of blocks on which they depend along the preceding diagonals (the diagonals $1, 2, \ldots, d-1$). Hence, the evaluation of blocks of a diagonal d can start after computing blocks of the last diagonal of blocks on which they depend, i.e. those of diagonal $(d-1)$.

However, Kechid and Myoupo (2009) have shown that the manner of computing the shortest path costs to nodes of a block allows starting before the end of the evaluation of blocks belonging to diagonal $(d-1)$. Thus, the evaluation of the blocks is done in a progressive fashion and starts as soon as possible. Theorem 3 indicates when computations of the shortest path costs to nodes of a block $SM(i, j)$ can be started.

Theorem 3 *After computing solutions of each diagonal h,* $\lceil (j-i)/2 \rceil + 1 \leq h \leq j - i + 1$*, at least two possible values of the shortest path from each node in block* $SM(i, j)$ *can be evaluated.*

Proof. To evaluate the shortest paths to nodes of a block $SM(i, j)$ of the diagonal $k = j - i + 1$, for each node, the cost of different edges and the value of the shortest path to the source node of each of these edges are required. $SM(i, j)$ receives edges from blocks $SM(i, j')$, such that $j' \in [i, j-1]$, and the costs of edges from a block $SM(i, j')$ can be evaluated as soon as the values of the shortest path of nodes of $SM(j', j)$ are computed (from Lemma 2). This is only possible from

Algorithm 6 CGM-based parallel algorithm of Kechid and Myoupo (2009) to solve the MPP

1: **for** $d = 1$ to p **do**
2: **Finalization** of computations of the shortest path costs to nodes in blocks belonging to the diagonal d;
3: **Communication** of block $SM(i, j)$ of current diagonal to processors that hold upper and right blocks;
4: **Update** the shortest path costs to each block belonging to diagonals $(d+1, d+2, \ldots, \min\{2\times(d-1), p\})$;

the diagonal t, such that $(t \geq j' - i + 1)$ and $(t \geq j - j' + 1)$. This implies that: $(t \geq \lceil j - i/2 \rceil + 1$. Thus, after evaluating the diagonal h, such that $t \leq h \leq j - i + 1$, the blocks $SM(i, h+i-1)$, $SM(h+i-1, j)$, $SM(i, j-h+1)$, and $SM(j-h+1, j)$ are evaluated. And thus, two possible values of the shortest path to each of nodes of $SM(i, j)$ are computable. Those whose incoming edges come from $SM(i, h+i-1)$ and $SM(i, j-h+1)$. ■

The CGM-based parallel algorithm of Kechid and Myoupo (2009) is a succession of p similar steps. The blocks are evaluated in parallel from the first diagonal of blocks, followed by the second, and so on till the last. The overall structure is given by Algorithm 6.

Evaluating the shortest path cost to a node of a block belonging to the diagonal d starts at the diagonal $\lceil d/2 \rceil$. After the computation of blocks on diagonal d (line 2 in Algorithm 6), each block is forwarded (line 3 in Algorithm 6) to processors that need these blocks for updating (line 4 in Algorithm 6) or for finalizing (line 2 in Algorithm 6) the computations of values in next steps. In fact, two tasks have to be done at step $(d+1)$:

1 - for each block belonging to the diagonal m, $(d+2 \leq m \leq 2d)$, some new values may be computed and an update of its temporary values is done (path relaxation principle (Cormen et al., 2009));

2 - some new values may be computed and a final update of its temporary values is done for each block of the diagonal $(d+1)$.

These processes are repeated at each step until step p where only the finalization phase is carried out. At step $d = 1$, only finalization and communication phases are executed. It is not difficult to notice that computing the shortest path costs (in which jumps from blocks $SM(i,k)$ are involved) to blocks $SM(i, j)$ is equivalent

Algorithm 7 Finalization phase using in Algorithm 6 to evaluate blocks $SM(i,j)$

1: **for** $d = (j-i-\theta(n,p))$ to $(j-i)$ **do**
2: **for each** node (a,b) of diagonal d belonging to $SM(i,j)$ **do**
3: $SP[a,b] \leftarrow \min\{SP[a,b]$, weight of paths whose final edge are jumps coming from block $SM(i,i+\theta(n,p))$, weights of paths whose final edges are internal jumps, weights of paths whose final edges are unit edges$\}$;

Algorithm 8 Updating phase using in Algorithm 6 to refresh the shortest path costs to nodes of blocks $SM(i,j)$

1: **for** $d = (j-i-\theta(n,p))$ to $(j-i)$ **do**
2: $M_1 \leftarrow$ matrix-multiplication$(+,\min)$ $(SM(i,h+i-1), SM(h+i-1,j))$;
3: $M_2 \leftarrow$ matrix-multiplication$(+,\min)$ $(SM(i,j-h+1), SM(j-h+1,j))$;
4: $SP[i,j] \leftarrow \min\{SP[i,j], M_1, M_2\}$;

to the sequential matrix-multiplication $(+,\min)$[2] of the two matrices $SM(i,k)$ and $SM(k-\theta(n,p)+2,j)$. Pseudocodes of finalization and updating phases in Algorithm 6 are given by Algorithms 7 and 8, respectively.

After the $(j-i)$th iteration of the above algorithms, the only paths which remain to evaluate for nodes in $SM(i,j)$ are those whose last edge is :

1 - either an unit edge (vertical or horizontal);

2 - or a horizontal jump, which comes from an internal node in $SM(i,j)$;

3 - or a horizontal jump, which comes from a node in $SM(i,i)$.

In any case, computing the weight induced by each of these paths (due to these edges) to a node (i',j') of $SM(i,j)$ needs the shortest path cost from a node (e',f') of $SM(i,j)$ such that $f'-e' < j'-i'$. In cases (1) and (2), this value is necessary to compute the shortest path cost of the start node of the last edge. In case (3), this value is necessary for computing the weight of the last edge. Therefore, Algorithm 7 is in fact a classical algorithm of the shortest path in $SM(i,j)$, in which each node can receive a simple edge or a jump from an internal node in the dynamic graph D'_n. In summary, Theorem 4 yields the complexity of the CGM-based parallel solution of Kechid and Myoupo (2009).

Theorem 4 *The CGM-based parallel solution of Kechid and Myoupo (2009) runs in $O\left(n^3/p\right)$ execution time with $O(p)$ communication rounds in the worst case.*

2. Matrix-multiplication $(+,min)$ is a matrix multiplication in which the multiplication and summation operations are replaced by addition and the minimum, respectively.

2.2.5 - CGM-based parallel solution of Kengne and Myoupo (2012)

This section presents the CGM-based parallel solution of Kengne and Myoupo (2012). They proposed a partitioning strategy that reduces the number of subgraphs (or blocks) of the dynamic graph D'_n as the number of processors increases to minimize the number of communication rounds. The dependencies between blocks are similar to those of Kechid and Myoupo (2009) since it is only the size of blocks that changes. Then, they propose a distribution scheme that fairly assigns blocks onto processors. This mapping balances better the loads between the processors than the alternative bidirectional projection mapping proposed by Kechid and Myoupo (2009) (described in Section 2.2.4). Moreover, it limits to at most two the number of blocks that a processor must evaluate. Nonetheless, the overall structure of their CGM-based parallel algorithm is similar to that of Kechid and Myoupo (2009).

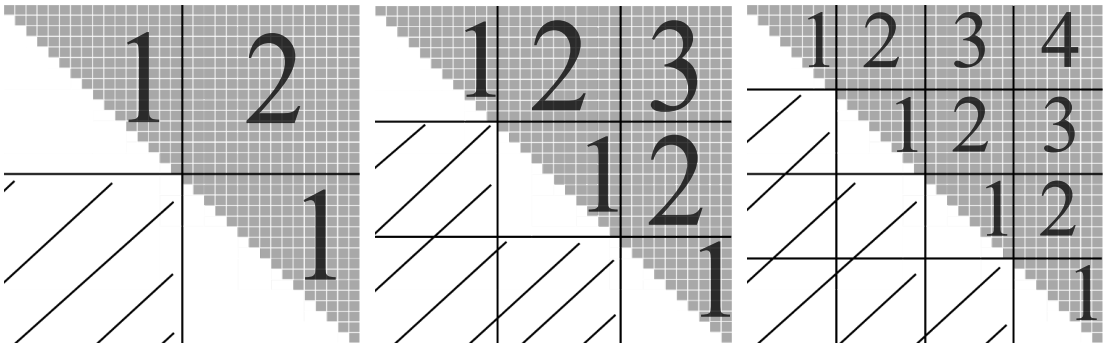
Dynamic graph partitioning

Kengne and Myoupo (2012) subdivide the shortest path matrix SP into $f(p)$ rows and $f(p)$ columns of blocks, where $f(p) = \lceil \sqrt{2p} \rceil$. A block $SM(i,j)$ is a $\theta(n,p) \times \theta(n,p)$ matrix, where $\theta(n,p) = \lceil n/f(p) \rceil$. The size of this block is larger than those obtained after applying the partitioning strategy of Kechid and Myoupo (2009) since $f(p) < p$ when $p > 2$. This means that entries of a block, illustrated by Equation (2.2), are more extensive and therefore, evaluating a block will require more computing time.

Figures 18a, 18b, and 18c show three scenarios of this partitioning for $n = 32$ and $p \in \{2,3,4,5,6,7,8\}$. SP is partitioned into $f(p)(f(p)+1)/2$ blocks. For example, in Figure 18b (respectively Figure 18c), SP is partitioned into six blocks (respectively ten blocks) when $p \in \{3,4\}$ (respectively $p \in \{5,6,7,8\}$). Usually, all blocks are not full when n mod $f(p) \neq 0$. For example in Figure 18b, while blocks in the first two columns are full, those in the third column are not. There are p blocks at the first diagonal and one block at the last. Recall that in these figures, the number in each block represents the diagonal in which it belongs.

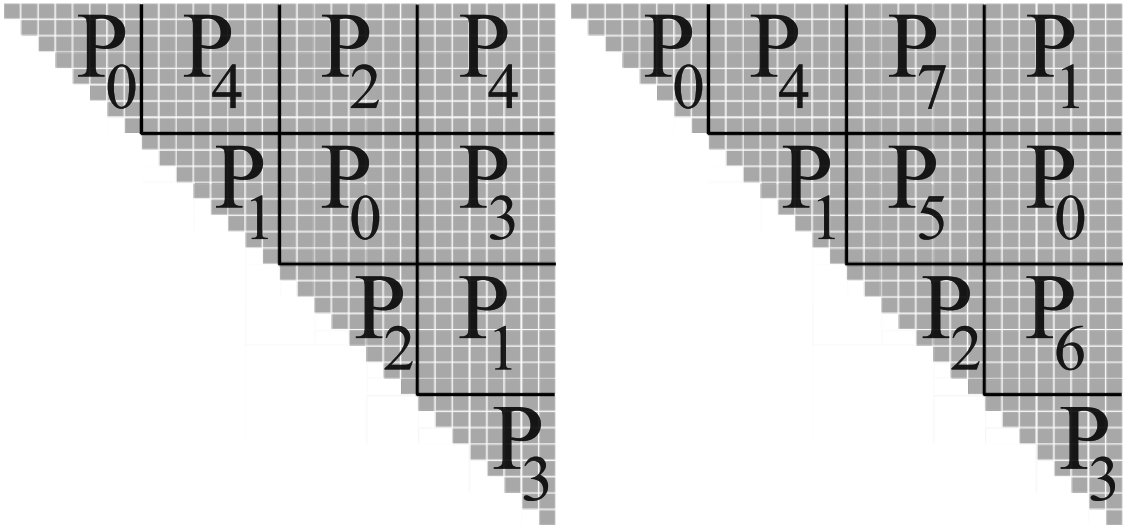
Mapping blocks onto processors

The mapping proposed by Kengne and Myoupo (2012) consists in assigning the blocks of a given diagonal from the upper leftmost corner to the lower rightmost corner. First, the blocks of the first diagonal are assigned from processor P_0 to processor $P_{f(p)-1}$. Then, the process is repeated on the next diagonal starting with processor $P_{f(p)}$, and so on until a block has been assigned to each processor. The mapping starts again with processor P_0, and continues along the diagonals with a



(a) $p = 2$ **(b)** $p \in \{3,4\}$ **(c)** $p \in \{5,6,7,8\}$

Figure 18 – *Shortest path matrix partitioning strategy proposed by Kengne and Myoupo (2012) for* $n = 32$ *and* $p \in \{2,3,4,5,6,7,8\}$



(a) $p = 5$ **(b)** $p = 8$

Figure 19 – *Snake-like mapping on five and eight processors*

snake-like path until the last diagonal $f(p)$. Figures 19a and 19b give an example of this mapping on five and eight processors.

The communication scheme corresponding to this mapping is simple and easy to implement like the alternative bidirectional projection mapping proposed by Kechid and Myoupo (2009). Moreover, this mapping allows processors to remain active as soon as possible and minimizes the number of communication rounds. It also ensures load balancing because it allows some processors to evaluate at most one block more than others. For example, Figure 19b shows that only processors P_0 and P_1 evaluate one more block than others. However, this mapping does not optimize communications since a processor does not usually hold the upper blocks that are on the same row and column as the block that has been assigned to that processor.

Algorithm 9 CGM-based parallel algorithm of Kengne and Myoupo (2012) to solve the MPP

1: **for** $d = 1$ to $f(p)$ **do**
2: **Finalization** of computations of the shortest path costs to nodes in blocks belonging to the diagonal d;
3: **Communication** of block $SM(i, j)$ of current diagonal to processors that hold upper and right blocks;
4: **Update** the shortest path costs to each block belonging to diagonals $(d+1, d+2, \ldots, \min\{2\times(d-1), f(p)\})$;

Lemma 3 *After partitioning the shortest path matrix into b submatrices (blocks) such that $p < b \leq 2p$ and applying the snake-like mapping, each processor evaluates at most two blocks.*

Proof. The shortest path matrix is partitioned into $b = f(p)(f(p)+1)/2$ blocks, where $f(p) = \lceil\sqrt{2p}\rceil$. It is deducible that $p < b \leq 2p$. Given that the snake-like mapping allows processors to evaluate at most one block more than others, each processor evaluates at most two blocks. ■

CGM-based parallel algorithm

The CGM-based parallel algorithm of Kengne and Myoupo (2012) is given by Algorithm 9. It consists of $f(p)$ similar steps as the one in Kechid and Myoupo (2009). As seen in Section 2.2.4, and according to Theorem 3, the updates for a block $SM(i, j)$ (which concern jumps from block $SM(i,k)$) are equivalent to a matrix multiplication $(+, \min)$ of matrices $SM(i,k)$ and $SM(k-\theta(n,p)+2, j)$. Evaluating a block of diagonal d starts as soon as the diagonal $\lceil d/2\rceil$ is evaluated, i.e. at step $(\lceil d/2\rceil + 1)$.

Theorem 5 *The CGM-based parallel solution of Kengne and Myoupo (2012) runs in $O\left(n^3/p\right)$ execution time with $\lceil\sqrt{2p}\rceil$ communication rounds in the worst case.*

2.3 - Optimal binary search tree problem

2.3.1 - Overview

A tree is a data structure that hierarchically represents a set of elements (often called keys) (Deo, 1974; Knuth, 1997). It consists of a set of vertices and edges. A node stores a key and an edge is a link between two vertices. A path is a sequence of

vertices in which two consecutive vertices are connected by an edge (Deo, 1974). Any two vertices in a tree are connected by a single path. In fact, a tree is an undirected acyclic connected graph (Knuth, 1997).

A rooted tree is a tree in which a vertex has been designated as the root (Knuth, 1997). Consider that there is a path from the root vertex to a vertex x. If there is a path from x to a vertex y, then x is an ancestor of y, and y is a descendant of x (Deo, 1974). If there are no intermediate vertices on the path between x and y, then x is a father of y, and y is a child of x (Deo, 1974). The root vertex is the only vertex that has no parent in a rooted tree. If there is no path between vertex x and another vertex, then x is a leaf (Deo, 1974).

A binary tree is a rooted tree where each vertex has at most two children: a left child and a right child (Deo, 1974; Knuth, 1997). The subtree of a vertex x in a tree is the part of that tree containing x, as well as its descendants. Thus, the left subtree (respectively the right subtree) of a vertex x is the subtree of the left child (respectively the right child) of x (Deo, 1974; Knuth, 1997).

A binary search tree is a binary tree that maintains a set of sorted keys according to the following rule: for each vertex x of the tree, the key of x is larger than all the keys contained in the left subtree of x and it is smaller than all the keys contained in the right subtree of x (Nagaraj, 1997). Figure 20 shows an example of a binary search tree corresponding to the set of keys $\langle$c, e, f, g, h, k, l, n, o, r, s$\rangle$ sorted in alphabetical order. A binary search tree is typically used to efficiently search for a particular key among a set of sorted keys. The cost of searching for this key is equal to length of the path from the root to this key plus one, which is in fact, the depth of this key (Cormen et al., 2009). For example, in Figure 20, the cost of searching for the key "n" (the root vertex) is equal to 1 and the cost of searching for the key "h" is equal to 5.

The overall cost of searching, which is equal to the sum of the cost of searching of each key, must be usually as less as possible in many applications. Self-balancing binary search trees[3] minimize this cost by ensuring that all keys are as near the root vertex as possible. However, when a query frequency is associated with each key, this tree will not be the most efficient in some cases. Indeed, it would be better to place the most frequently queried keys closer to the root vertex, and to place the less frequently queried keys further from the root vertex. For example, consider the set of sorted keys $\langle$a, b, c$\rangle$ with respective frequencies of 3, 1, and 7. The cost of searching for a key x is equal to the depth of x multiplied by

3. A self-balancing binary search tree is a binary search tree that automatically attempts to keep its height (maximal number of levels below the root vertex) as small as possible at all times (Knuth, 1998).

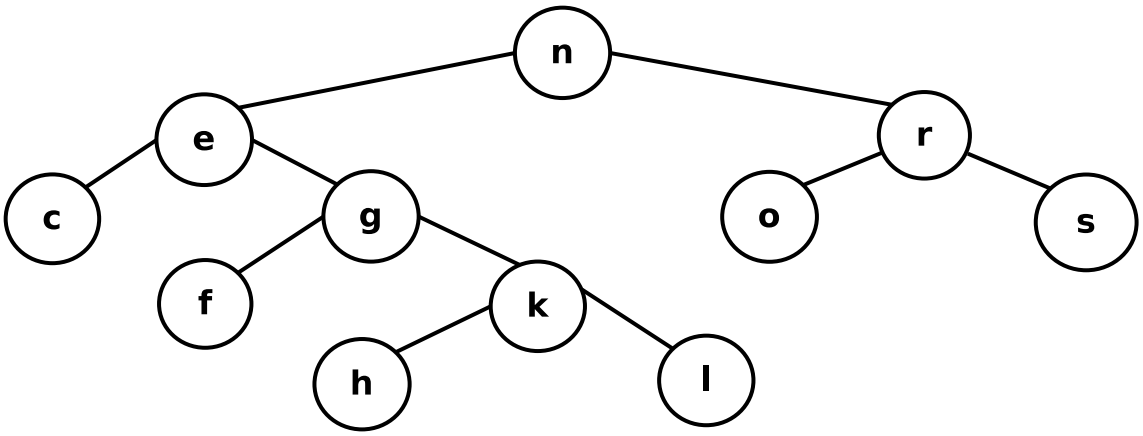


Figure 20 – *Example of a binary search tree corresponding to the set of letters ⟨c, e, f, g, h, k, l, n, o, r, s⟩ sorted in alphabetical order*

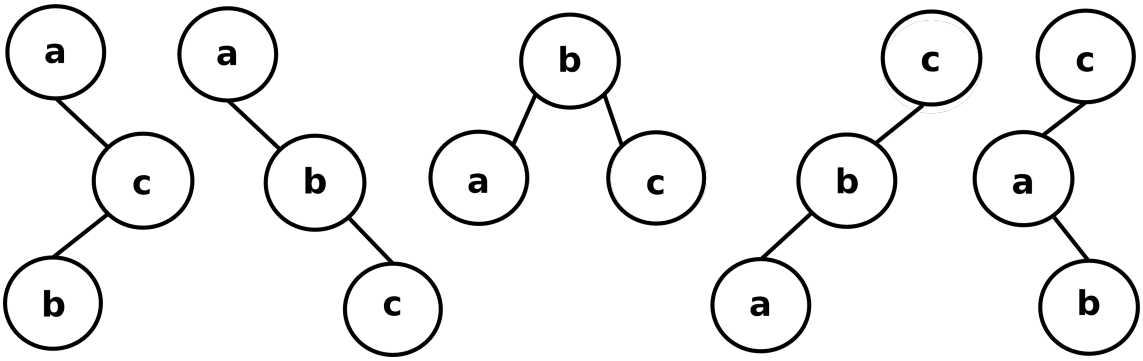


Figure 21 – *Five possible binary search trees obtained from the set of sorted keys ⟨a, b, c⟩*

the frequency of x. Of the five possible binary search trees, depicted in Figure 21, that can be obtained from these keys, the tree that minimizes the overall cost of searching is determined as follows:

- for the first tree, it is equal to $(1 \times 3) + (2 \times 7) + (3 \times 1) = 20$;
- for the second tree, it is equal to $(1 \times 3) + (2 \times 1) + (3 \times 7) = 26$;
- for the third tree, it is equal to $(1 \times 1) + (2 \times 3) + (2 \times 7) = 21$;
- for the fourth tree, it is equal to $(1 \times 7) + (2 \times 1) + (3 \times 3) = 18$;
- for the fifth tree, it is equal to $(1 \times 7) + (2 \times 3) + (3 \times 1) = 16$.

As shown in Figure 21, the third binary search tree is balanced but the fifth is not. Nevertheless, the overall search cost of the latter is lower than the self-balanced binary search tree (the third tree in Figure 21) and is the lowest of all; thus, it is the optimal binary search tree for the above example.

More formally, the optimal binary search tree (OBST) problem can be defined as follows: consider a set of n sorted keys $K = \langle k_1, k_2, \ldots, k_n \rangle$, such that $k_1 < k_2 < \cdots < k_n$. The probability of finding a key k_i is denoted by p_i. In the case where the searched element is not in K, consider a set of $(n+1)$ dummy keys $D = \langle d_0, d_1, \ldots, d_n \rangle$. Indeed, d_0 represents the set of values that are smaller than k_1 and d_n the set of values that are larger than k_n. A dummy key d_i, such that $1 \leq i < n$, represents the set of values between k_i and k_{i+1}. The probability of finding a dummy key d_i is denoted by q_i. In summary, either the search ends in success (i.e. finding the searched key k_i) or in failure (i.e. finding a dummy key d_i); thus the sum of probabilities of success and failure is equal to 1. A binary search tree T constructed from these keys is composed of n internal vertices in the set K and $(n+1)$ leaves in the set D. Equation (2.3) gives the overall cost of searching of T:

$$Cost(T) = \sum_{i=1}^{n} (depth(k_i) \times p_i) + \sum_{i=0}^{n} (depth(d_i) \times q_i) \tag{2.3}$$

where $depth(x)$ is the depth of the key x. The OBST problem consists in finding the binary search tree that will have the lowest overall cost of searching.

Let $w(i,j) = p_{i+1} + \cdots + p_j + q_i + q_{i+1} + \cdots + q_j$. The minimum cost of searching of a tree $T_{i,j}$ from the set of keys $K_{i,j} = \langle k_{i+1}, k_{i+2}, \ldots, k_j \rangle$ and the set of dummy keys $D_{i,j} = \langle d_i, d_{i+1}, \ldots, d_j \rangle$ is denoted by $Tree[i,j]$ and defined by:

$$Tree[i,j] = \begin{cases} q_i & \text{if } 0 \leq i = j \leq n, \\ \min\limits_{i \leq k < j} \{Tree[i,k] + Tree[k+1,j] + w(i,j)\} & \text{if } 0 \leq i < j \leq n. \end{cases} \tag{2.4}$$

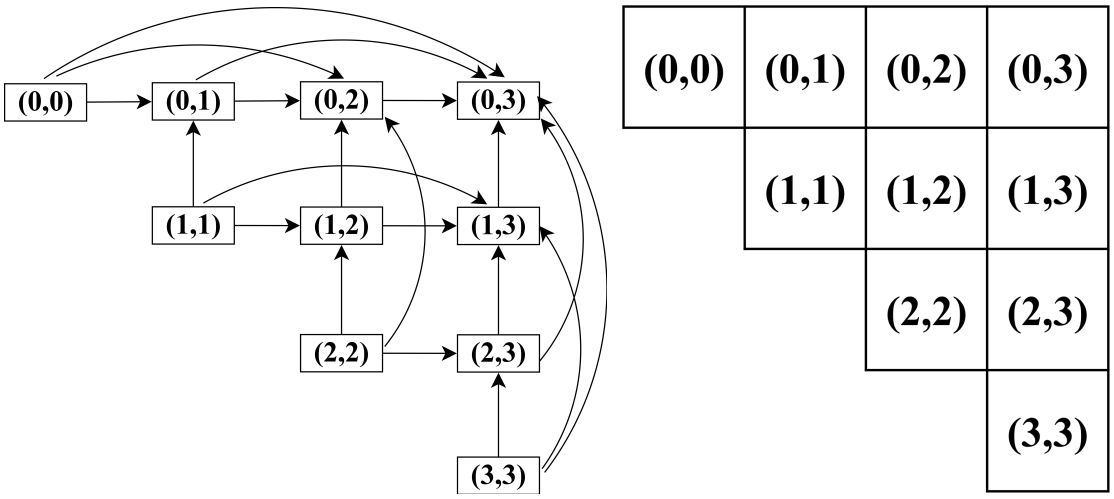
Finding the optimal binary search tree from the set of keys K and the set of dummy keys D is reduced to compute $Tree[0,n]$. Equation (2.4) is equivalent to Equation (1.8) as $Opt = \min$, $Init(i) = q_i$, and $F(i,k,j) = w(i,j)$, where $0 \leq i \leq k \leq j \leq n$.

The straightforward sequential algorithm of Godbole (1973), running in $O\left(n^3\right)$ time and $O\left(n^2\right)$ space, is given by Algorithm 10. The DP table (named *Tree* in Algorithm 10) stores the value of the optimal cost of searching of $T_{i,j}$ (see line 10 in Algorithm 10). The tracking table (named *Cut* in Algorithm 10) stores the value of the index k, which minimizes $Tree[i,j]$ (see line 11 in Algorithm 10). It is the optimal decomposition value of $T_{i,j}$ into two subtrees. For a problem of size $n = 3$, the DAG and the DP table are illustrated in Figures 22a and 22b, respectively.

Applications of optimal search binary trees are numerous. One of the most obvious is the search for a word in a dictionary. Indeed, from the words of a dictionary and the access frequency of each word, an optimal binary search tree can be built to quickly and efficiently answer to a query. An application derived from the latter

Algorithm 10 Sequential algorithm of Godbole (1973) to solve the OBST problem

```
1:  for i = 0 to n do
2:      Tree[i,i] ← q_i;
3:  for d = 1 to n do
4:      for i = 0 to n − d + 1 do
5:          j ← n − d + 1;
6:          Tree[i, j] ← ∞;
7:          for k = i to j − 1 do
8:              c ← Tree[i,k] + Tree[k + 1, j] + w(i, j);
9:              if c < Tree[i, j] then
10:                 Tree[i, j] ← c;
11:                 Cut[i, j] ← k;
```



(a) *Task graph* **(b)** *Dynamic-programming table*

Figure 22 – *Task graph and the dynamic-programming table used to compute* $Tree[0,3]$

is the translation of a word from one language to another (Cormen et al., 2009). El-Qawasmeh (2004) used an optimal binary search tree to solve the word prediction problem. This problem attempts to guess and update the next word in a sentence as it is typed. In view of these different applications, an optimal binary search tree should be constructed from the sequential algorithm of Knuth (1971), which is an improved version of the sequential algorithm of Godbole (1973), running in $O\left(n^2\right)$ time and space.

2.3.2 - Sequential algorithm of Knuth (1971)

Knuth (1971) noticed entries of tracking table *Cut* satisfy the *monotonicity property* such that $Cut[i, j-1] \leq Cut[i, j] \leq Cut[i+1, j]$ for all $0 \leq i < j \leq n$. This property means that computing $Cut[i, j]$ consists in finding all indices between $Cut[i, j-1]$

Algorithm 11 Sequential algorithm of Knuth (1971) to solve the OBST problem

```
1:  for i = 0 to n do
2:      Tree[i,i] ← q_i;
3:  for d = 1 to n do
4:      for i = 0 to n − d + 1 do
5:          j ← n − d + 1;
6:          Tree[i, j] ← ∞;
7:          for k = Cut[i, j − 1] to Cut[i + 1, j] do
8:              c ← Tree[i,k] + Tree[k + 1, j] + w(i, j);
9:              if c < Tree[i, j] then
10:                 Tree[i, j] ← c;
11:                 Cut[i, j] ← k;
```

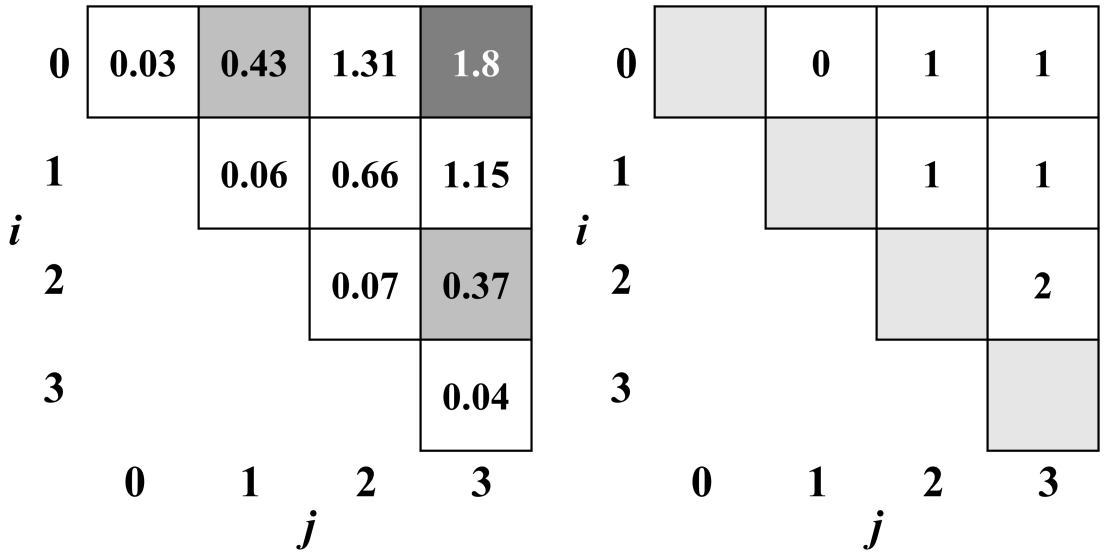
Table 1 – *Example of probabilities of three sorted keys*

i	0	1	2	3
p_i		0.25	0.4	0.15
q_i	0.03	0.06	0.07	0.04

and $Cut[i+1, j]$, instead of between i and j as it is done in line 7 of Algorithm 10. Knuth (1971) thus achieves to evaluate the $\Theta\left(n^2\right)$ subproblems in constant time. Algorithm 11 draws a big picture.

Parallelization constraint of the sequential algorithm of Knuth (1971)

One downside to the speedup of Knuth (1971), however, is that the number of comparison operations for entries in the same diagonal varies from one entry to another compared to the classical version. Consider for example the set of three sorted keys with probabilities given in Table 1. Figures 23a and 23b depict, respectively, the DP table and the tracking table that have been filled while determining the optimal binary search tree from these probabilities. Figure 23a shows that the value of $Tree[0,3]$ is computed only from the optimal solutions of the pair of subproblems $((0,1),(2,3))$, since $Cut[0,2]$ and $Cut[1,3]$ are equal to 1. Yet in the classical version, the value of $Tree[0,3]$ should be computed from the optimal solutions of the pairs of subproblems $((0,0),(1,3))$, $((0,1),(2,3))$, and $((0,2),(3,3))$. Therefore, when designing parallel algorithms based on the sequential algorithm of Knuth (1971), there is no guarantee that the processors will have the same load if entries on a diagonal are fairly distributed among them, as shown in Figures 19a and 19b for example. It is the main parallelization constraint of this algorithm.



(a) *Dynamic-programming table* **(b)** *Tracking table*

Figure 23 – *Dynamic-programming and tracking tables filled while determining the optimal binary search binary tree from probabilities of three sorted keys given in Table 1*

Literature review on the parallelization of the sequential algorithm of Knuth (1971)

Although parallelization of the sequential algorithm of Godbole (1973) has been widely studied in the literature on different parallel computing models (see Section 2.2.2), little work has been done on parallelization of the sequential algorithm of Knuth (1971). On the PRAM model, Karpinski and Rytter (1994) designed a sublinear time parallel algorithm to construct optimal binary search trees. It requires $O\left(n^{1-\varepsilon}\log n\right)$ computation time with the total work $O\left(n^{2+2\varepsilon}\right)$ for an arbitrarily small constant $0 < \varepsilon \leq 0.5$. Recall that the total work is equal to the computation time multiplied by the number of processors (Karpinski and Rytter, 1994). On CREW-PRAM machines, Karpinski et al. (1996) presented an algorithm running in $O\left(n^{0.6}\right)$ computation time with n processors. On realistic models of parallel machines, Craus (2002) proposed an $O(n)$-time parallel algorithm using $O\left(n^2\right)$ processors on very-large-scale integration (VLSI) architectures. Wani and Ahmad (2019) proposed a parallel implementation of the sequential algorithm of Knuth (1971) on GPU architectures. For a problem of size 16384, they achieved a speedup factor of 409 on a NVIDIA GeForce GTX 570 chip and a speedup factor of 745 on a NVIDIA GeForce GTX 1060 chip. On shared-memory architectures, after demonstrating that dependencies of subproblems available in the code implementing the sequential algorithm of Knuth (1971) allow generating only 2D tiled code, using the polyhedral model, Bielecki et al. (2021) proposed a way of trans-

forming this algorithm to a modified one exposing dependencies to generate 3D parallel tiled code. Experimentations conducted using OpenMP demonstrated that the 3D tiled code considerably outperforms the 2D tiled code.

On distributed-memory architectures, Kechid and Myoupo (2008a) proposed the first CGM-based parallel solution to solve the OBST problem. It requires $O\left(n^2/p\right)$ execution time with $O\left(p\right)$ communication rounds. They improved this solution in their works done in (Kechid and Myoupo, 2009) to solve the MPP. Myoupo and Kengne (2014b) built a CGM-based parallel solution, from their proposed solution in (Kengne and Myoupo, 2012) to solve the MPP, running in $O\left(n^2/\sqrt{p}\right)$ execution time with $\left\lceil\sqrt{2p}\right\rceil$ communication rounds. This solution uses the same partitioning strategy and the same distribution scheme strategy; except that the evaluation of blocks is not done in a progressive manner. Indeed, Myoupo and Kengne (2014b) showed that the progressive evaluation is not suitable because of the speedup of Knuth (1971). Later on, Kengne et al. (2016) highlighted the existence of a relationship between the execution time, the load balancing, and the number of communication rounds in a family of CGM-based parallel solutions solving the OBST problem. They proposed a general methodology for deriving a CGM-based parallel algorithm in accordance with the user's parameters. This solution is described more succinctly in Section 2.3.3.

2.3.3 - CGM-based parallel solution of Kengne et al. (2016)

Looking at the performance of CGM-based parallel solutions proposed by Kechid and Myoupo (2008a) and Myoupo and Kengne (2014b) to solve the OBST problem (and the solutions proposed by Kechid and Myoupo (2009) and Kengne and Myoupo (2012) to solve the MPP), Kengne et al. (2016) noticed that the execution time was related to the load balancing and the number of communication rounds. These criteria depend on the partitioning strategy and the distribution scheme strategy used when designing the CGM-based parallel solutions:

1 - When the dependency graph is subdivided into small-size blocks (Kechid and Myoupo, 2008a, 2009), the load difference between processors is small if one processor has one more block than another. However, the number of communication rounds will be high.

2 - When the dependency graph is subdivided into large-size blocks (Kengne and Myoupo, 2012; Myoupo and Kengne, 2014b), the number of communication rounds of the corresponding algorithm is reduced since there are few blocks. However, the load of processors will be unbalanced.

These criteria in fact have a significant impact on the global communication time, which in turn has an impact on the total execution time. Indeed, when the blocks are small, the number of communication rounds is high since there are many blocks. Thus, excessive communication will lead to communication overhead, which will deteriorate the global communication time. On the other hand, when the blocks are large, the evaluation of a block will take a long time because of the load imbalance. So a processor that is waiting for this block to start or continue its computation will wait longer to receive this block. As a result, Kengne et al. (2016) showed that these criteria are contradictory because it is difficult to simultaneously optimize them in a CGM-based parallel solution.

Kengne et al. (2016) proposed a CGM-based parallel solution that gives the end-user the choice to optimize one criterion according to their own goal. They generalized the ideas of dependency graph partitioning and distribution scheme introduced in (Kechid and Myoupo, 2008a, 2009; Kengne and Myoupo, 2012; Myoupo and Kengne, 2014b). The end-user derives a CGM-based parallel algorithm from these parameters:

- g (the granularity) : it is used in the partitioning strategy to define the size and number of blocks;
- p (the number of processors) : it is used in the distribution scheme strategy to assign blocks onto processors.

The complexity of the derived CGM-based parallel algorithm depends on these parameters. Kengne et al. (2016) also proposed a simulator based on the work of Fotso et al. (2010), which allows users to modify these parameters and observe the performance (number of blocks, number of communication rounds, load balancing, and efficiency) of the derived CGM-based parallel algorithm.

Dependency graph partitioning

Kengne et al. (2016) partition the DAG into $g(g+1)$ blocks. A block $SM(i,j)$ is a matrix of size $\theta(n,g) \times \theta(n,g)$, where $\theta(n,g) = \lceil (n+1)/g \rceil$. This strategy encompasses all the previous ones because :

- it is equivalent to Kechid and Myoupo (2008a) when $g = 2p$;
- it is equivalent to Kechid and Myoupo (2009) when $g = p$;
- it is equivalent to Myoupo and Kengne (2014b) when $g = \lceil \sqrt{2p} \rceil$.

It is straightforward to notice that g can be larger, smaller or equal to p. Nevertheless, the blocks obtained after the partitioning must be distributed fairly to processors whatever the value of g.

Blocks' dependency analysis

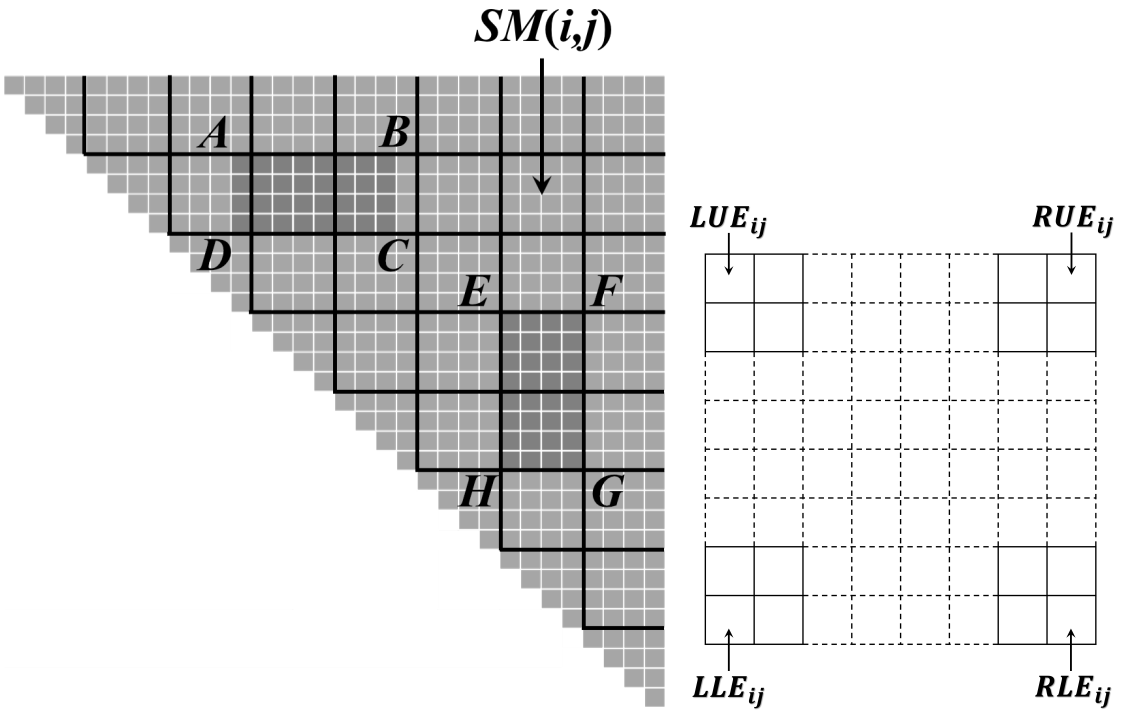
Figures 24a and 24b respectively show an example of dependencies and extremities of a block $SM(i,j)$ after applying the partitioning strategy of Kengne et al. (2016). The extremities of $SM(i,j)$ are defined by :

- the leftmost upper entry $LUE_{ij} = (i, j-\theta(n,g)+1)$;
- the rightmost upper entry $RUE_{ij} = (i,j)$;
- the leftmost lower entry $LLE_{ij} = (i+\theta(n,g)-1, j-\theta(n,g)+1)$;
- the rightmost lower entry $RLE_{ij} = (i+\theta(n,g)-1, j)$.

Figure 24a illustrates eight points (A, B, C, D, E, F, G, and H) that identify blocks on which the block $SM(i,j)$ depends :

- $A = Tree\left[LUE_{i,j-3\times\theta(n,g)-1}\right]$
- $B = Tree\left[LUE_{i,j-\theta(n,g)-2}\right]$
- $C = Tree\left[LLE_{i,j-\theta(n,g)-2}\right]$
- $D = Tree\left[LLE_{i,j-3\times\theta(n,g)-1}\right]$
- $E = Tree\left[LLE_{i+\theta(n,g)+1,j}\right]$
- $F = Tree\left[RLE_{i+\theta(n,g)+1,j}\right]$
- $G = Tree\left[RLE_{i+3\times\theta(n,g),j}\right]$
- $H = Tree\left[LLE_{i+3\times\theta(n,g),j}\right]$

All lower blocks that are in the same row and column as the block $SM(i,j)$ are no longer absolutely required to evaluate $SM(i,j)$ as a consequence of the speedup of Knuth (1971) (Kengne, 2014). In Figure 24a for example, only the five most shaded blocks will be needed. In other cases, it could have been just four, three or two blocks. Moreover, it may happen that some nodes of blocks located in extremities are needed instead of the whole blocks. It depends on the input data. Therefore, the speedup of Knuth (1971) does not allow estimating the exact load of a block before its evaluation (Kengne, 2014).



(a) *Dependencies of* $SM(i,j)$ **(b)** *Extremities of* $SM(i,j)$

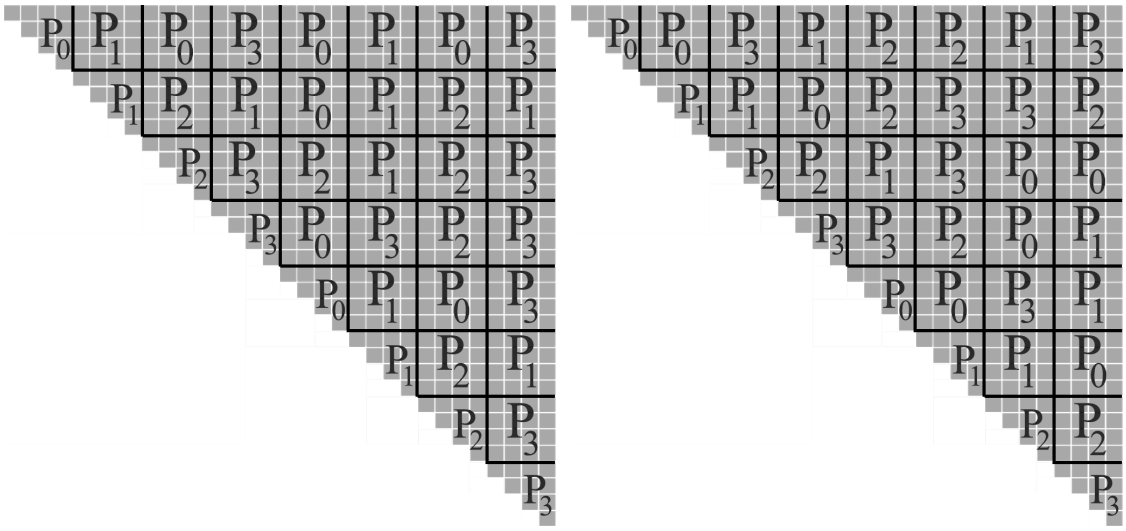
Figure 24 – *Dependencies and extremities of a block* $SM(i,j)$ *after applying the partitioning strategy of Kengne et al. (2016)*

However, the blocks on which the block $SM(i,j)$ depends are not located on the same diagonal as $SM(i,j)$. Consequently, they can be carried out in parallel.

Mapping blocks onto processors

Kengne et al. (2016) implemented the alternative bidirectional projection mapping (described in Section 2.2.4, page 55) and snake-like mapping (described in Section 2.2.5, page 60). They extended these strategies and let the end-user choose one of them according to their own goal.

The alternative bidirectional projection mapping is not practical when $g < p$ because no blocks will be assigned to processors $P_g, P_{g+1}, \ldots, P_{p-1}$; these processors will remain idle throughout the problem-solving process. It is only practical when $g = k \times p$, where k is a positive integer. k blocks of the first diagonal are assigned to a processor. The blocks on other diagonals are then projected onto the first diagonal. Figure 25a shows an example of this mapping when $g = 8$ and $p = 4$. This mapping minimizes communications and makes the processors less idle compared to Figure 17b when $g = p = 8$. Indeed, a processor has more than half of the blocks in the row and in the column to which it has been assigned its first blocks of the first diagonal. Furthermore, in Figure 25a, the first processor (processor P_2)



(a) *Alternative bidirectional projection mapping* **(b)** *Snake-like mapping*

Figure 25 – *Alternative bidirectional projection mapping and snake-like mapping on four processors when* $g = 8$

becomes idle after step 6; yet in Figure 17b, it is after step 4 that the first processor (processor P_4) becomes idle.

The snake-like mapping is convenient whatever the value of g. When $g \geq p$, the blocks of the first diagonal are assigned from processor P_0 to processor P_{p-1}. Then, the process is repeated starting again with processor P_0 until all blocks of the first diagonal are assigned (when $g > p$), continues to the next diagonal and so on until a block has been assigned to each processor. Figure 25b gives an example of this mapping when $g = 8$ and $p = 4$. This mapping does not minimize communications enough compared to alternative bidirectional projection mapping. However, Kengne et al. (2016) have shown that the snake-like mapping is more efficient and balances the load of processors better than alternative bidirectional projection mapping.

CGM-based parallel algorithm

The analysis of the dependency of blocks showed that the evaluation of a block does not always depend on all lower blocks that are in the same row and column as this block. Evaluating blocks in a progressive fashion, like the MPP (see Section 2.2.4, page 57), would not be ideal because some unnecessary computations could be performed (Kengne, 2014). Kengne et al. (2016) evaluated blocks in a non-progressive fashion, i.e. the evaluation of blocks of the diagonal d starts after computing blocks of the diagonal $(d-1)$.

Algorithm 12 CGM-based parallel algorithm of Kengne et al. (2016) to solve the OBST problem

1: **for** $d = 1$ to R **do**
2: **Computation** of blocks belonging to the round d using Algorithm 11;
3: **Communication** of entries (*Tree* and *Cut* tables) required for computing each block of rounds $\{d+1, d+2, \ldots, R\}$;

Lemma 4 states the number of communication rounds of the CGM-based parallel algorithm of Kengne et al. (2016). It is equal to g when $g \leq p$ because each diagonal of blocks contains at most p blocks; this means that each processor will evaluate at most one block per diagonal. This is not the case when $g > p$. In fact, a processor will evaluate at most k blocks per diagonal when $g = k \times p$. For example in Figures 25a and 25b, a processor evaluates at most two blocks from diagonal 1 to diagonal 4; and there is 12 communication rounds. Considering R as the number of communication rounds regardless of the values of g and p, Algorithm 12 gives an overview of the CGM-based parallel algorithm of Kengne et al. (2016).

Lemma 4 *The number of communication rounds of the CGM-based parallel algorithm of Kengne et al. (2016) is defined by:*

$$R = \begin{cases} g & \text{if } g \leq p, \\ \left\lceil \dfrac{g(g+1) - p(p-1)}{2p} \right\rceil + p & \text{if } g > p. \end{cases}$$

Proof. When $g \leq p$, it is clear that there is g communication rounds. When $g > p$, diagonals that contain at least p blocks are evaluated in $\left\lceil \frac{g(g+1)-p(p-1)}{2p} \right\rceil$ communication rounds. The last $(p-1)$ diagonal of blocks are evaluated in $(p-1)$ rounds. The potential blocks located in the diagonal that contains p blocks and has not yet been considered can be evaluated in at most one round. ■

Theorem 6 *The CGM-based parallel solution of Kengne et al. (2016) runs in* $O\left(n^2 \times R/g^2\right)$ *execution time with R communication rounds in the worst case.*

Simulator of Kengne et al. (2016)

Kengne et al. (2016) proposed a simulator to allow users to modify the parameters g and p and observe the performance of the derived CGM-based parallel algorithm. From the partitioning strategy and the distribution scheme strategy, they easily deduce the size and number of blocks, and the number of communication rounds. They relied on the work of Fotso et al. (2010), which computes the load of blocks

to infer the load imbalance of processors and the efficiency of the derived CGM-based parallel algorithm. Since the speedup of Knuth (1971) makes it impossible to predict the exact load of a block, they used the maximum load, i.e. the one obtained with the sequential algorithm of Godbole (1973).

From Equation (1.8) described in Section 1.8, the evaluation of each entry of each diagonal d requires $3(d-1)$ arithmetic operations; since these $(d-1)$ treatments consist of two addition operations and a minimum operation (Fotso et al., 2010). Consider that the size of a block is $m \times m$, where $m = \theta(n,g) = \lceil (n+1)/g \rceil$. Blocks of the first diagonal will induce a load of

$$\alpha = 3 \times \sum_{k=1}^{m} k\,(m-k) = 3 \times \frac{(m-1) \times m \times (m+1)}{6}$$

and blocks of a diagonal $d > 1$ will induce a load of

$$3 \times \left(\sum_{k=1}^{m} k\,((i-2)\,m+k) + \sum_{k=1}^{m-1} k\,(im-k) \right) = 3 \times (i-1)\,m^3.$$

Denoting by $\beta = 3 \times m^3$, the workload induced by the evaluation of a block of the first diagonal is α, while those of diagonals i, such that $i > 1$, are $(i-1)\beta$. From these values, they estimated the load imbalance of processors and predicted the efficiency of the algorithm. Recall that the load imbalance of processors is the load difference between the processor that performs most of the work and the one that performs the least. For example, if the end-user chooses the alternative bidirectional projection mapping with $g = 8$ and $p = 4$ (illustrated in Figure 25a), then processor P_2 will have the smallest load $(2\alpha + 16\beta)$ and processor P_3 will have the largest $(2\alpha + 26\beta)$. The load imbalance in this case will equal to 10β. On the other hand, if he chooses the snake-like mapping illustrated in Figure 25b, the load imbalance will equal to 6β since processor P_0 will have a load of $(2\alpha + 18\beta)$ and processor P_3 a load of $(2\alpha + 24\beta)$.

The main shortcoming of this simulator is that it can provide inaccurate load balancing metrics to the end-user. In fact, as mentioned earlier, this simulator takes into account the maximum load of blocks to estimate the load of processors. However, it is not possible to know the exact load of a block because it varies according to the input data due to the speedup of Knuth (1971). This shortcoming could mislead the end-user when running the derived CGM-based parallel algorithm on supercomputers.

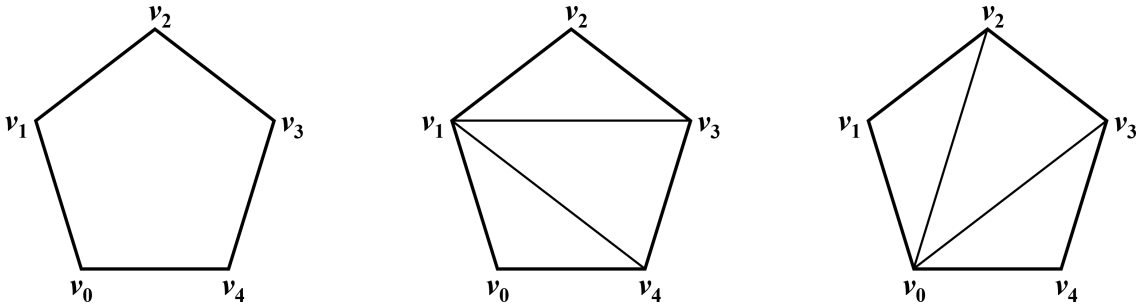


Figure 26 – *A convex polygon and two different triangulations*

2.4 - Triangulation of a convex polygon problem

2.4.1 - Overview

Computational geometry is the branch of computer science that studies algorithms for geometrical problems. Its application areas include computer graphics, robotics, geographic information systems, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, statistics, and many others. A computational geometry problem usually takes as input a description of one or more geometric shapes, such as a set of line segments or points. It returns a response to a query such as the number of points of intersection, or perhaps a new geometric object (Cormen et al., 2009).

A polygon P is a geometric object that has a structure bordered by connections between vertices $\langle v_0, v_1, \ldots, v_n \rangle$. It is made up of $(n+1)$ side edges $v_i v_j$ such that $j = (i+1) \bmod (n+1)$. Any line segment connecting two vertices v_i and v_j such that $j > (i+1)$ is called *a chord*. It splits the polygon into two smaller polygons. A polygon is said to be convex when any chord $v_i v_j$ passes either through the side edge or through the inside of the polygon, but never through the outside of the polygon[4] (Bradford, 1994). A triangulation of a convex polygon (TCP) can be thought of as a set of chords that divide the polygon into triangles such that two chords never intersect except at a vertex. Figure 26 shows a convex polygon $P = \langle v_0, v_1, v_2, v_3, v_4 \rangle$ and two different ways to triangulate it.

In a weighted convex polygon P, a positive integer weight $w(v)$ is assigned to each vertex v. The cost of a triangle $v_i v_j v_k$ of P is the product of the weights of its vertices, $w(v_i) \times w(v_j) \times w(v_k)$. The cost of a TCP is the sum of the costs of its triangles. The problem is to find an optimal triangulation that minimizes

4. A chord always divides a convex polygon into two convex polygons (Bradford, 1994).

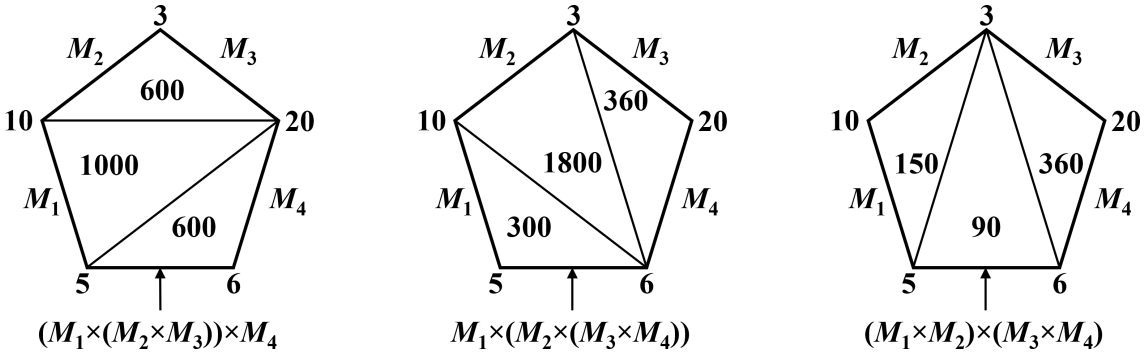


Figure 27 – *Different ways of triangulating the convex polygon $P = \langle 5,10,3,20,6\rangle$ and the corresponding parenthesis of the product of four matrices M_1, M_2, M_3, and M_4*

the cost of P. Denoting by $T[i,j]$ the optimal triangulation cost of a sub-polygon $P_{i,j} = \langle v_{i-1}, v_i, \ldots, v_j\rangle$, this problem can be modeled by Equation (2.5) :

$$T[i,j] = \begin{cases} 0 & \text{if } 1 \leq i = j \leq n, \\ \min\limits_{i \leq k < j} \{T[i,k] + T[k+1,j] + w(v_{i-1}) \times w(v_k) \times w(v_j)\} & \text{if } 1 \leq i < j \leq n. \end{cases} \tag{2.5}$$

It is easy to see that this equation is equivalent to Equation (2.1) in Section 2.2.1. In fact, this problem is closely related to the MCOP (Hu and Shing, 1982, 1984; Yao, 1982). Consider n matrices $M_1 \times M_2 \times \cdots \times M_i \times \cdots \times M_n$ where a matrix M_i has dimensions $(d_{i-1} \times d_i)$. A $(n+1)$-sided convex polygon P can be constructed with the matrices dimensions $\langle d_0, d_1, \ldots, d_n\rangle$ such that a matrix M_i corresponds to a side edge $d_{i-1}d_i$ and a product $M_i \times M_{i+1} \times \cdots \times M_j$ corresponds to a chord $d_i d_j$. Thus, there exists a one-to-one correspondence between the different ways of parenthesizing n matrices and the possible ways of triangulating a $(n+1)$-sided convex polygon (Hu and Shing, 1982, 1984). An example is illustrated in Figure 27. It is deducible that a solution to one remains relevant to the other; so the sequential algorithm of Godbole (1973) is relevant to the TCP problem. However, Yao (1982) improved the sequential algorithm of Godbole (1973) based on the *quadrangle inequality* property. It runs in $O\left(n^2\right)$ time and space.

2.4.2 - Sequential algorithm of Yao (1982)

Recursive solution

Consider now that the vertices of a convex polygon P are ordered. A particular ordering is chosen and remains fixed during the triangulation process when some weights are equal. A vertex at the rank j is designated by w_j. So the convex polygon $P = \langle 5,10,3,20,6\rangle$ of Figure 27 can be represented by $\langle w_2, w_4, w_1, w_5, w_3\rangle$. A sub-polygon $P_{i,j}$ of P consists of those vertices lying between w_i and w_j in a clock-

Algorithm 13 Recursif algorithm of Yao (1982) to solve the TCP problem

```
1:  function PARTITION(P)
2:      if |P| ≤ 2 then
3:          return ∅;
4:      else if P is a triangle then
5:          return P;
6:      else if w1 and w2 are not adjacent then
7:          return PARTITION(P1,2) ∪ PARTITION(P2,1);
8:      else if w1 and w3 are not adjacent then
9:          return PARTITION(P1,3) ∪ PARTITION(P3,1);
10:     else
11:         return better of {PARTITION(P2,3) ∪ PARTITION(P3,2),
                              PARTITION(P1,4) ∪ PARTITION(P4,1)};
```

wise traversal. For example in Figure 27, $P_{2,1} = \langle w_2, w_4, w_1 \rangle$, $P_{4,3} = \langle w_4, w_1, w_5, w_3 \rangle$, and $P_{3,4} = \langle w_3, w_2, w_4 \rangle$. The term *partition* will also be used to refer to a triangulation.

Starting from a quadrilateral, Yao (1982) brought out some properties that led to Lemma 5 and set up Algorithm 13 to find an optimal triangulation from this lemma.

Lemma 5 *Given a convex polygon P, there exists an optimal partition for which the following is true :*

1 - w_1 and w_2 are adjacent either by a side edge or by a chord, similarly for w_1 and w_3;

2 - if both w_1w_2 and w_1w_3 are side edges, then either w_1w_4 or w_2w_3 exists as a chord.

As the last clause "else" (line 10 in Algorithm 13) could generate two problems of size $n-c$ for a small constant c in the worst case, this algorithm requires an exponential time. Yao (1982) reduces this complexity to $O\left(n^2\right)$ time and space by proposing a dynamic-programming solution divided into two steps :

1 - identify and organize the $O\left(n^2\right)$ distinct sub-polygons (subproblems) into a multi-level DAG;

2 - solve these subproblems in a bottom-up manner as organized in step 1.

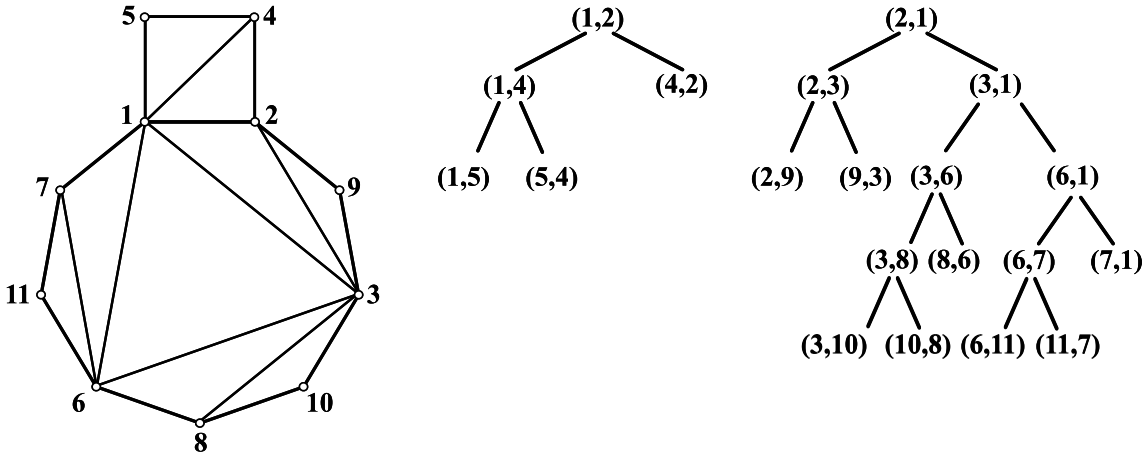


Figure 28 – *A convex polygon P and the corresponding DAG*

Dynamic-programming solution

In step 1, the constructed DAG is a forest consisting of two binary trees in which roots are designated by nodes w_1w_2 and w_2w_1. Each binary tree corresponds to sub-polygons $P_{1,2}$ and $P_{2,1}$ such that $P = P_{1,2} \cup P_{2,1}$. This DAG is structured as follows:

1 - a nonleaf node w_iw_j is a chord of P such that all vertices w_k in $P_{i,j}$ fulfill $k \geq \max\{i, j\}$;

2 - any nonleaf node w_iw_j has exactly two child nodes, namely w_iw_k and w_kw_j where k is the smallest index (aside from i and j) in $P_{i,j}$;

3 - the leaves are the side edges of P;

4 - an order relation $\prec$ is defined on the set of nodes such that $w_{i'}w_{j'} \prec w_iw_j$ implies $P_{i',j'} \subseteq P_{i,j}$.

By scanning the weights of P in a clockwise direction, Yao (1982) identifies and generates the nodes of the DAG in a slightly modified postorder traversal. Figure 28 gives an example of a convex polygon P and the corresponding DAG. Nodes in this graph are generated in the following order: w_1w_5, w_5w_4, w_1w_4, w_4w_2, w_3w_{10}, $w_{10}w_8$, w_3w_8, w_8w_6, w_6w_{11}, $w_{11}w_7$, w_6w_7, w_7w_1, w_2w_9, w_9w_3, w_3w_6, w_6w_1, w_2w_3, w_3w_1, w_1w_2, w_2w_1.

A sub-polygon Q of P is called *a cone*, if $Q = P_{i,j} \cup w_iw_jw_k$ where $b = w_iw_j$ is a chord or a side edge of P, and $k \leq \min\{i, j\}$. A cone Q is also designated by $(b, w_k\rangle$. The existing partial orders on the nodes of the DAG and on the weights of P induce a natural alphabetic order on cones. In fact, a cone $Q' = (b', w_{k'}\rangle$ precedes a cone $Q = (b, w_k\rangle$ if either $b' \prec b$, or $b' = b$, and $k' \geq k$ (Yao, 1982).

Algorithm 14 Dynamic-programming algorithm of Yao (1982) to solve the TCP problem

1: **for each** $b = w_i w_j \in B$ **do** ▷ B is the ordered set of nodes and consider that $i < j$
2: **if** b is a leaf **then**
3: **for all** cones $Q = (b, w_k\rangle$ such that $k \leq i$ **do**
4: **if** $w_k = w_i$ **then**
5: Partition $[Q] \leftarrow \emptyset$; ▷ $Q = w_i w_j$
6: **else**
7: Partition $[Q] \leftarrow Q$; ▷ $Q = w_i w_j w_k$
8: **if** b is not a leaf **then**
9: **for all** cones $Q = (b, w_k\rangle$ such that $k \leq i$ **do**
10: **if** $w_k = w_i$ **then**
11: Partition $[Q] \leftarrow$ Partition $[(\text{leftChild}(b), w_i\rangle] \cup$ Partition $[(\text{rightChild}(b), w_i\rangle]$;
12: **else**
13: Partition $[Q] \leftarrow$ better of $\{$Partition $[(b, w_i\rangle] \cup w_i w_j w_k$, Partition $[(\text{leftChild}(b), w_k\rangle] \cup$ Partition $[(\text{rightChild}(b), w_k\rangle]\}$;
14: **return** Partition $[P_{1,2}] \cup$ Partition $[P_{2,1}]$;

Lemma 6 *For a cone $Q = (b, w_k\rangle$, any sub-polygon that may arise in the execution of Partition$[Q]$ is either a triangle, or a cone Q' which precedes Q.*

Thus, step 2 of the dynamic-programming algorithm of Yao (1982) consists of computing and tabulating the solutions to all cones in accordance with their precedence order. Indeed, evaluating a node $b = w_i w_j$ of the DAG consists in computing all the cones $Q = (b, w_k\rangle$ such that $k \leq i < j$. Algorithm 14 draws the big picture. Partition$[Q]$ refers to the DP table containing the optimal solution to cone Q. The outer for loop (line 1 in Algorithm 14) iterates over all b in the order they are generated in step 1. The inner for loops (lines 3 and 9 in Algorithm 14) iterate over all w_k with $k \leq i$ in decreasing order. As there are at most $2n$ nodes in the DAG, and at most n cones to evaluate for a given node, this algorithm runs in $O\left(n^2\right)$ time and space.

Enhancement of Myoupo and Kengne (2014a) : additional preprocessing routine

When evaluating a node $w_i w_j$ in Algorithm 14, the vertices of the polygon to be considered are all w_k such that $k \leq i$. This evaluation is made in decreasing order of these weights. Yao (1982) does not explain how to effectively identify these vertices, nor how to obtain them in decreasing order. Thus, the most natural way to look for these vertices w_k is to scan the entire set of vertices located between w_i and w_j and compare their weights to w_i. However, some of them may have

weights greater than w_i. For example in Figure 28, to evaluate the node w_3w_6, the set of candidate vertices $\langle w_{11}, w_7, w_1, w_5, w_4, w_2, w_9, w_3 \rangle$ must be entirely scanned in order to select only three : w_3, w_2, and w_1. This leads to a huge waste of time because in this case five of the eight candidate vertices are not useful.

To solve this problem, Myoupo and Kengne (2014a) proposed an additional preprocessing routine to the sequential algorithm of Yao (1982). It consists in sorting the weights of the polygon vertices, through the quicksort algorithm, before evaluating the DAG nodes. Thus, the evaluation of the node w_iw_j starts from the vertex w_i since it is the vertex with the highest weight. It continues in decreasing order in the sorted weight set of the polygon vertices until the vertex with the lowest weight. In this way, Myoupo and Kengne (2014a) identify more easily the set of vertices whose weights are less than or equal to w_i. Experimental results showed that their proposal had a significant improvement over the sequential algorithm of Yao (1982) in term of execution time. However, it did not change the time complexity.

Parallelization constraint of the sequential algorithm of Yao (1982)

In contrast to the sequential algorithm of Godbole (1973) where the dependency graph is the same whatever the dimensions of matrices to be parenthesized, the DAG corresponding to the sequential algorithm of Yao (1982) is different for two products of matrices of the same length whose dimensions are not the same. Since this DAG is not known beforehand, a classical design of a CGM-based parallel solution based on the sequential algorithm of Yao (1982), like the sequential algorithm of Godbole (1973) for the MPP or the OBST problem, is not possible (Kechid and Myoupo, 2008b; Myoupo and Kengne, 2014a). Thus, Kechid and Myoupo (2008b) automated the dependency graph partitioning and the distribution scheme for each instance of dimensions of matrices.

2.4.3 - CGM-based parallel solution of Kechid and Myoupo (2008b)

Kechid and Myoupo (2008b) focused on parallelizing step 2 of the $O\left(n^2\right)$-time dynamic-programming solution of Yao (1982). They did not find it relevant to parallelize step 1 running in $O(n)$ time since in practice $n^2/p \gg n$. Algorithm 15 gives an overview of the CGM-based parallel solution of Kechid and Myoupo (2008b) (it is the same as the one of Myoupo and Kengne (2014a) described in Section 2.4.4). This solution is broken down into three steps:

Algorithm 15 CGM-based parallel solution of Kechid and Myoupo (2008b) to solve the TCP problem

1: **Construction** of the DAG;
2: **Partitioning** of the DAG into subgraphs and **mapping** onto processors;
3: **Evaluation** of the subgraphs using Algorithm 14;

1 - First, each processor performs step 1 of the sequential algorithm of Yao (1982), i.e. the generation of nodes of the DAG.

2 - Then, each processor runs an algorithm called *dynamic design algorithm*. This algorithm consists of dividing the DAG into subgraphs called *bands*, then subdividing each band into sub-bands called *blocks*, and finally distributing each block onto processors. It ensures that blocks of the same band are independent of each other so that they can be processed in parallel, and that they have the same size to facilitate load balancing during the mapping.

3 - Finally, processors evaluate the blocks assigned to them in parallel through a succession of computation and communication rounds.

Dependency graph partitioning

Kechid and Myoupo (2008b) partition the DAG into S bands from the global load of the DAG obtained by adding up the smallest indexes of the DAG nodes. They denote by B_i the ith band, $i \in \{1,2,\ldots,S\}$, and by T_{cb} the maximum load of a band. Each band B_i is then divided into NB_i blocks. Algorithm 16 shows an overview of the dependency graph partitioning algorithm of Kechid and Myoupo (2008b). It determines the best value of the triplet (S, T_{cb}, NB_i) that minimizes the overall computational load per processor expressed by Equation (2.6):

$$cost = \sum_{i=1}^{S} (NB_i/p) \times T_{cb} \tag{2.6}$$

To produce the partitioning corresponding to a given T_{cb}, Algorithm 16 traverses in a bottom-up fashion the original DAG (produced by step 1 of the sequential solution of Yao (1982)) from leaves to build bands one-by-one by making successive calls to a *band-building* function (line 11 in Algorithm 16), which takes as input the current DAG and T_{cb}, builds a band and returns the number of blocks of this band. On the first call to this function, the current DAG corresponds to the original DAG. Then, before each call of this function, the band built at the previous call

Algorithm 16 Dependency graph partitioning algorithm of Kechid and Myoupo (2008b)

1: $T_c \leftarrow 0$;
2: **for each** $w_i w_j \in B$ **do** ▷ B is the ordered set of nodes
3: $T_c \leftarrow T_c + \min\{i, j\}$;
4: $min\_cost \leftarrow +\infty$;
5: **for** $\alpha = 2$ to p **do**
6: $T_{cb} \leftarrow T_c/\alpha p$;
7: $S \leftarrow 0$;
8: $i \leftarrow 1$;
9: $B' \leftarrow B$;
10: **while** $B' \neq \emptyset$ **do**
11: $NB_i \leftarrow$ building the band B_i from B' and T_{cb};
12: $S \leftarrow S + 1$;
13: $B' \leftarrow B' \setminus \{B_i\}$;
14: $i \leftarrow i + 1$;
15: $current\_cost \leftarrow$ cost of the current partitioning computed by Equation (2.6);
16: **if** $current\_cost < min\_cost$ **then**
17: $min\_cost \leftarrow current\_cost$;
18: $T_{cb_{optimal}} \leftarrow T_{cb}$;
19: **return** the partitioning corresponding to $T_{cb_{optimal}}$;

is truncated from the current DAG (line 13 in Algorithm 16). When the current DAG becomes empty, the partitioning is completed. Kechid and Myoupo (2008b) repeat this process $(p-1)$ times to find the best value of the triplet (S, T_{cb}, NB_i) that minimizes the cost given by Equation (2.6).

Lemma 7 *The dependency graph partitioning algorithm of Kechid and Myoupo (2008b) requires $O(n \times p)$ execution time.*

Proof. Nodes in the DAG are traversed $(p-1)$ times to seek the partitioning that minimizes the cost in Equation (2.6). ■

Blocks' dependency analysis

Since the evaluation of a DAG node (i, j) depends only on the already computed values of its two child nodes (i,k) and (k, j), the values communicated by processors are those of the root nodes of blocks they hold; and these nodes are not plentiful. Moreover, this information is used only by processors holding the father nodes of the root nodes of blocks. In other words, the values of the root nodes of blocks in the band B_i are therefore only communicated to processors that own blocks in the band B_{i+1}.

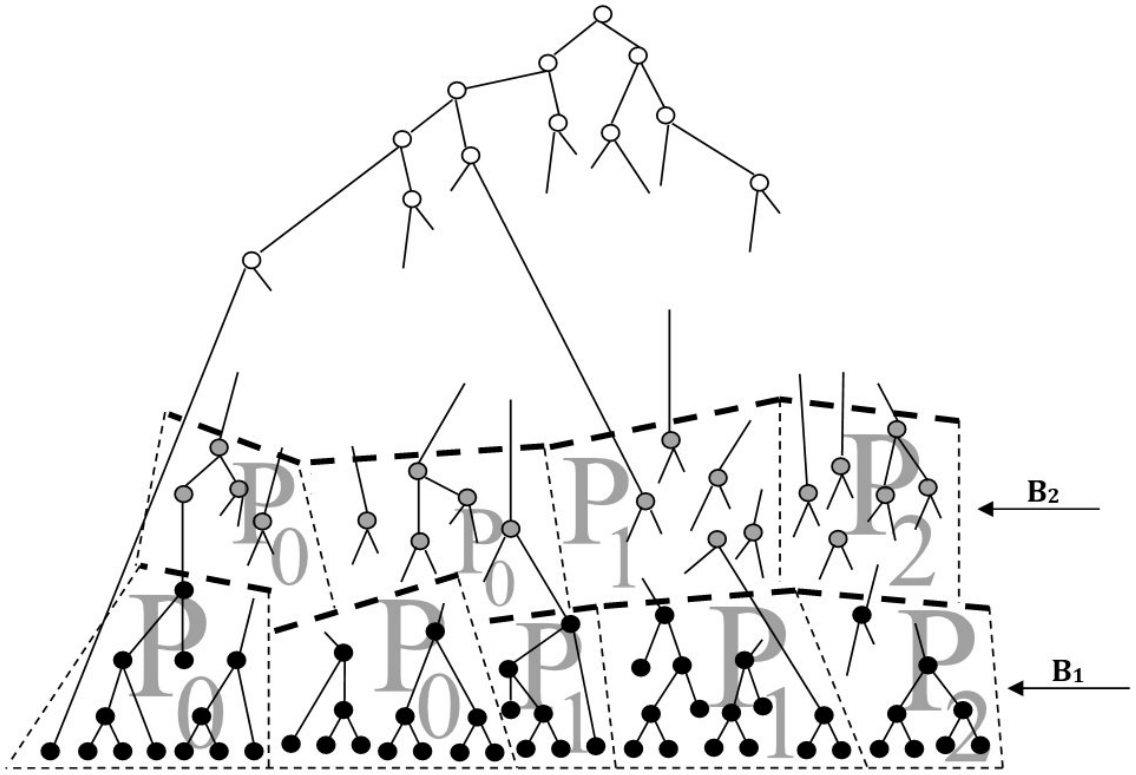


Figure 29 – *Distribution scheme of blocks on three processors proposed by Kechid and Myoupo (2008b)*

With such dependencies, the communication time will be very low. Contrary to all the CGM-based parallel solutions presented so far, in this one, the local computation time is the main part of the total execution time.

Mapping blocks onto processors

Once the DAG is divided into bands and blocks, Kechid and Myoupo (2008b) distribute the blocks to processors while ensuring good load balancing. The NB_i blocks in the band B_i are assigned to processors as follows:

- When $NB_i \geq p$, each processor evaluates $\lceil NB_i/p \rceil$ blocks. This mapping ensures that for each band B_i, blocks assigned to a processor are contiguous to minimize the amount of data to be communicated.
- When $NB_i < p$, $(NB_i - p)$ processors will be idle in the computation round i. To manage this idleness, Kechid and Myoupo (2008b) uses the snake-like mapping (described in Section 2.2.5, page 60).

Figure 29 depicts an example of this mapping on three processors. Nodes colored in black are those of the first band, those colored in gray are those of the second band, and uncolored nodes are those of the other bands.

Theorem 7 *In the worst case, the complexity of the CGM-based parallel solution of Kechid and Myoupo (2008b) to solve the TCP problem, given in terms of the BSP cost model, is expressed by:*

$$S \times L + S \times g \times O\left(n^2/p\right) + O\left(n \times p\right) + \sum_{i=1}^{S} \left(NB_i/p\right) \times T_{cb_{optimal}}$$

2.4.4 - CGM-based parallel solution of Myoupo and Kengne (2014a)

Since the dynamic design algorithm (line 2 in Algorithm 15) is executed sequentially on each processor, it must be carried out with minimum time. However, the solution proposed by Kechid and Myoupo (2008b), running in $O\left(n \times p\right)$ time to parallelize the $O\left(n^2\right)$-time sequential algorithm of Yao (1982), will be time intensive when the number of processors increases (although $n \gg p$). This is not interesting because this algorithm must be scalable to the increase of the number of processors.

Myoupo and Kengne (2014a) improve this solution by making the dynamic design algorithm less dependent on the number of processors. The strength of their solution comes from the fact that processors will handle more than half of the problem without communicating. Indeed, the DAG has at most $2n$ nodes among which $(n+1)$ leaves. Since these leaves are all independent, they can be evaluated in parallel. Moreover, more than half of the computational load is induced by these leaves because the evaluation of each internal node requires less computation than that of a given leaf since the computational load induced by an internal node is smaller than that of each of its two children, and there are more leaves than internal nodes. Thus, if the first band B_1 contains all leaves of the DAG, then it will induce a computational load greater than half the global load of the DAG. If $NB_1 \geq p$, then each processor will evaluate at least one block of the first band. Myoupo and Kengne (2014a) propose two algorithms for dependency graph partitioning: one focused on load balancing and the other focused on minimizing the number of communication rounds. They also improved the distribution scheme proposed by Kechid and Myoupo (2008b). Their mapping ensures that processors communicate blocks after computing to start the evaluation of some blocks as soon as possible.

Dependency graph partitioning balancing the load of processors

To balance the load between processors, the size of blocks must be small. So if some processors have one more block than others, as blocks have a relatively small load, the load difference will be reasonable. The downside of this approach is

Algorithm 17 Dependency graph partitioning algorithm of Myoupo and Kengne (2014a) balancing the load of processors

```
1: T_max ← 0;
2: for each w_i w_k and w_k w_j ∈ B do                ▷ B is the ordered set of nodes
3:     if w_i w_k is a leaf then
4:         T_max ← T_max + min{i,k};
5:     if w_k w_j is a leaf then
6:         T_max ← T_max + min{k,j};
7:     if w_i w_k and w_k w_j are leaves then
8:         T_max ← T_max + min{i,j};
9: T_cb ← T_max/p;
10: S ← 0;
11: i ← 1;
12: B' ← B;
13: while B' ≠ ∅ do
14:     building the band B_i from B' and T_cb;
15:     S ← S + 1;
16:     B' ← B' \ {B_i};
17:     i ← i + 1;
18: return the partitioned DAG;
```

the growth of the number of blocks, which induce more communication between processors and thus increase the global communication time.

The principle of this strategy is the following: if (i,k) and (k,j) are two leaves of the DAG, then their father (i,j) can be part of the first band. Indeed, all internal nodes whose two children are leaves are all independent, and thus can be evaluated in parallel. Denoting by T_{max} the sum of loads of these internal nodes and that of all the leaves, Myoupo and Kengne (2014a) show that the best load of blocks of the first band is T_{max}/p. By keeping this load for blocks of next bands, they limit to two the number of blocks to be mapped onto a processor. Thus, a processor evaluates either two blocks on the first band B_1 (if $NB_1 > p$) and no blocks on the other bands, or one block on B_1 and possibly one block on all bands $\{2,\ldots,S\}$. Algorithm 17 draws the big picture.

Lemma 8 *The dependency graph partitioning algorithm of Myoupo and Kengne (2014a) balancing the load of processors runs in $O(n)$ execution time.*

Proof. This algorithm requires just two traverses of the DAG nodes : one to compute T_{max} and another to construct the S bands. ■

Dependency graph partitioning minimizing the number of communication rounds

The blocks should be as large as possible to minimize the number of communication rounds. In this way, the number of blocks is minimized and the processors exchange little information. The drawback of this approach is the possibility of a serious load imbalance between processors if they do not have the same number of blocks.

The principle of this strategy is to find the load of the largest first band that contains at least p blocks by changing the load of blocks. Formally, Myoupo and Kengne (2014a) assume that the number of blocks of the first band is greater than or equal to p when $T_{cb} = T_c/2p$ to vary T_{cb} in the search interval $\left[\frac{T_c}{2p}, \ldots, \frac{T_c}{p}\right]$. If T_c/p produces less than p blocks in the first band, they consider the median value $\frac{1}{2}\left(\frac{T_c}{p} + \frac{T_c}{2p}\right) = \frac{3T_c}{4p}$. They use each time using the median value between the largest known good value of T_{cb} (lower bound of the interval) and the smallest known bad value (upper bound of the interval) to optimize the result. The process stops when the search interval reaches the threshold T_c/p^{k+1}, where k is a positive integer; and the lower bound of the search interval is chosen. Thus, Myoupo and Kengne (2014a) obtain the load of the largest first band corresponding to the chosen threshold. Denoting by T_{int} the load of this band, the bands of the DAG are partitioned into blocks with a load of T_{int}/p. Algorithm 18 gives an overview of this strategy.

Lemma 9 *The dependency graph partitioning algorithm of Myoupo and Kengne (2014a) that minimizes the number of communication rounds requires $O(n \log p)$ execution time.*

Proof. The initial distance between the bounds of the interval in which the best value of T_{cb} is found is $T_c/p - T_c/2p = T_c/p$. This distance is divided by two until it reaches the threshold T_c/p^{k+1}. Thus, the number of divisions of this interval is the largest value of m, such that $T_c/2^m p \geq T_c/p^{k+1}$, i.e. $p^k \geq 2^m$ where $k \times \ln p \geq m \ln 2$ and thus $m \leq k \times \ln p / \ln 2$.

For each value of T_{cb}, all nodes of the DAG are traversed to compute the corresponding value of NB_1. Thus, $O(n \log p)$ execution time are required to determine the final value of T_{cb}. After the construction of the first band, a single traversal of the rest of the graph is performed for the construction of the other bands. ■

Mapping blocks onto processors

Myoupo and Kengne (2014a) are based on the distribution scheme proposed by Kechid and Myoupo (2008b). They are interested on the consecutive assignment

Algorithm 18 Dependency graph partitioning algorithm of Myoupo and Kengne (2014a) minimizing the number of communication rounds

```
1: T_c ← 0;
2: for each w_i w_j ∈ B do                          ▷ B is the ordered set of nodes
3:     T_c ← T_c + min{i, j};
4: sup ← T_c/p;
5: inf ← T_c/2p;
6: gap ← ⌈sup − inf⌉;
7: T_cb ← sup;
8: while gap ≥ T_c/p^2 do
9:     NB_1 ← 0;
10:    B' ← B;
11:    while no end band do
12:        current_block ← building a block from B' and T_cb;
13:        NB_1 ← NB_1 + 1;
14:        B' ← B' \ {current_block};
15:    if NB_1 ≥ p then
16:        inf ← T_cb;
17:    else
18:        sup ← T_cb;
19:    gap ← ⌈sup − inf⌉;
20:    T_cb ← ⌈(sup + inf)/2⌉;
21: B' ← B \ {B_1};
22: T_cb ← inf;
23: S ← 1;
24: i ← 2;
25: while B' ≠ ∅ do
26:    building the band B_i from B' and T_cb;
27:    S ← S + 1;
28:    B' ← B' \ {B_i};
29:    i ← i + 1;
30: return the partitioned DAG;
```

of blocks of a band B_i to a processor, which minimizes the amount of data to be communicated during communication rounds. Indeed, this strategy does not allow processors that hold the blocks of the band B_{i+1} to start the evaluation of their blocks as soon as possible. To solve this problem, Myoupo and Kengne (2014a) first assigns the first p blocks of the band B_1 to p processors, then they assign the rest of the $(NB_1 - p)$ blocks to processors $P_0, \ldots, P_{(NB_1-p)}$. Blocks of the band B_2 are assigned to processors starting with processor $P_{(NB_1-p+1)}$. This process is applied until the last band S.

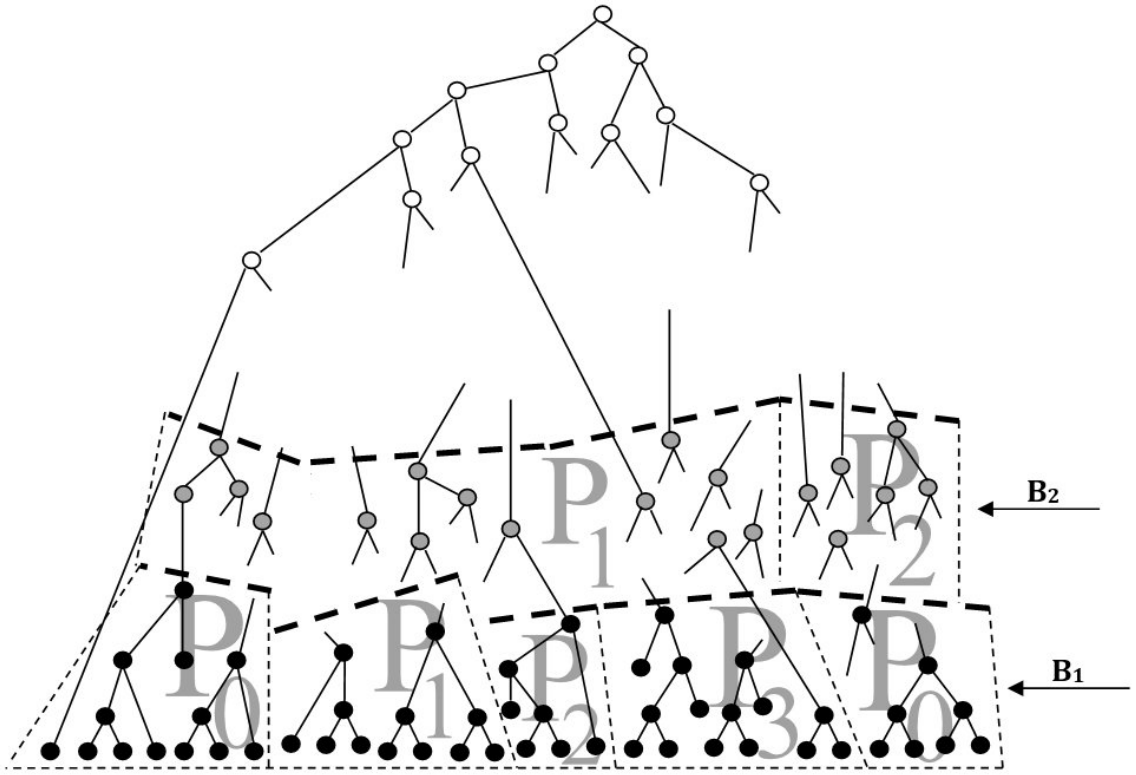


Figure 30 – *Distribution scheme of blocks on four processors proposed by Myoupo and Kengne (2014a)*

So, after the evaluation of the first p blocks of the band B_1, the values of the root nodes are communicated to processors owning blocks of the band B_2, which can immediately start the evaluation of their blocks. At the same time, the $(NB_1 - p)$ processors with an additional block on band B_1 can evaluate their second block. Figure 30 illustrates an example of this mapping on four processors.

Theorem 8 *In the worst case, to solve the TCP problem, the CGM-based parallel solution of Myoupo and Kengne (2014a) runs in* $O\left(n^2 \times S/p\right)$ *execution time with* S *communication rounds, such that* $2 \leq S < p$.

2.4.5 - Drawbacks of sequential and CGM-based parallel solutions

The trade-off of minimizing the number of communication rounds and balancing the load of processors still arises despite the CGM-based parallel solutions of Kechid and Myoupo (2008b) and Myoupo and Kengne (2014a). Moreover, for this problem, the computation time represents the largest part of the execution time. Despite the improved sequential algorithm of Myoupo and Kengne (2014a), an important observation was not made on how to evaluate the nodes of the DAG.

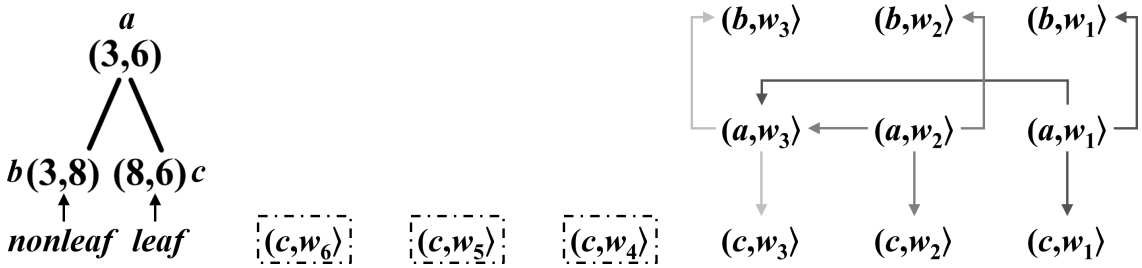


Figure 31 – *Evaluation of the nonleaf node $a = w_3w_6$ and dependencies between its cones $(a,w_3\rangle$, $(a,w_2\rangle$, and $(a,w_1\rangle$ to be computed*

In fact, neither Yao (1982), nor Kechid and Myoupo (2008b), nor Myoupo and Kengne (2014a) were interested in the dependencies between cones. They focused only on dependencies between nodes. Their solutions have the same drawback. It is about the unnecessary computation of some cones when evaluating a given node in the DAG. Figure 31 illustrates dependencies between cones of the nonleaf node $a = w_3w_6$ (see Figure 28) with its two sons $b = w_3w_8$ and $c = w_8w_6$. Let assume that all cones of nodes b and c are already computed. It is easy to see that the cones $(c,w_6\rangle$, $(c,w_5\rangle$, and $(c,w_4\rangle$ (which are in the boxes in Figure 31) will not be useful for the evaluation of the node a. Indeed, according to the definition of a given node w_iw_j with its children w_iw_k and w_kw_j such that $i < j < k$, there will always be $(j-i)$ computed cones that will not be necessary for the evaluation of the node w_iw_j and therefore, disregarding this fact can lead to a significant waste of time. The solution proposed in this thesis consists in organizing the evaluation of cones following their dependencies to avoid these unnecessary computations. Section 2.4.6 describes it more precisely.

2.4.6 - Our fast sequential algorithm

Consider a node $b = w_iw_j$ such that $i < j$. All the descendants $a = w_{i'}w_{j'}$ of the node b must compute a cone $Q = (a,w_i\rangle$. It is easy to see from the dynamic-programming equation (lines 10-13 in Algorithm 14) that each descendant a must first process the cone $Q' = (a,w_{i'}\rangle$ to compute Q. Consider that B is the ordered set of nodes in the DAG. By going from B in a top-down manner, i.e. from the last node w_2w_1 to the first, a stack $\delta_b = \{w_k\}_{k\leq i}$ can be built for each node b. A new DAG, in which each node b is associated to its stack δ_b, is obtained. Thus, his evaluation will consist in computing all cones $Q = (b,w_k\rangle$ such that $k \leq i$ and $w_k \in \delta_b$. The first cone $Q = (b,w_k\rangle$ to process is needed to compute the other cones $Q' = (b,w_{k'}\rangle$ such that $k' < k$. Depending on the nature of b, its stack δ_b will be constructed as follows:

Algorithm 19 Building the stack of every node in the DAG

```
1:  B' ← the reversed B set;                          ▷ B is the ordered set of nodes in the DAG
2:  push (δ_{w2w1}, w1);
3:  push (δ_{w1w2}, w1);
4:  for each b = wi wj ∈ B' do                        ▷ consider that i < j
5:      if b is not a leaf then
6:          δ_{leftChild(b)} ← δb;
7:          if leftChild(b) is not a leaf and top(δ_{leftChild(b)}) ≠ wi then
8:              push(δ_{leftChild(b)}, wi);
9:          δ_{rightChild(b)} ← δb;
10:         if rightChild(b) is not a leaf and top(δ_{rightChild(b)}) ≠ wi then
11:             push(δ_{rightChild(b)}, wi);
```

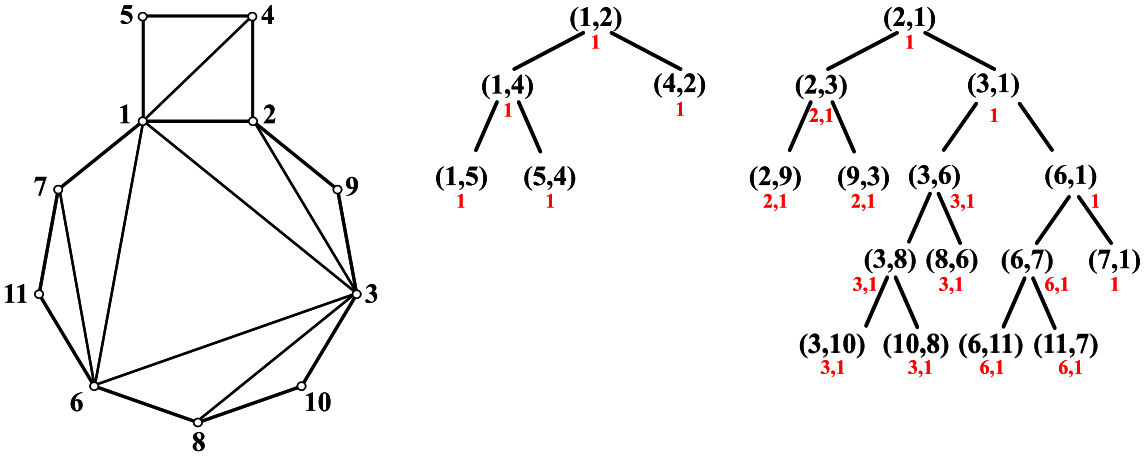


Figure 32 – *A convex polygon P and the new DAG obtained after building the stack of every node. Elements of the stack are colored red. The leftmost vertex corresponds to the top of the stack*

1 - if b is a leaf or not, then assign $\delta_{b'}$ to δ_b, where b' is the father of b;

2 - if b is not a leaf, then push the vertex w_i onto δ_b if $\text{top}(\delta_b) \neq w_i$. Recall that the function `top` returns the top of the stack.

Algorithm 19 gives an overview. In the beginning, the vertex w_1 is pushed in the stacks of the last two nodes w_1w_2 and w_2w_1 since they are the root nodes of the DAG. Next, for each node b, δ_b is built according to its nature. This algorithm requires $O(n)$ execution time. Consider again the convex polygon P presented in Figure 28. Figure 32 shows the new DAG obtained after applying Algorithm 19. The overall number of operations to be performed, obtained by adding up the smallest indexes of the DAG nodes, is 62. However, when the stack size of every node in the DAG is added up, it is halved.

Our dynamic-programming solution is then divided into three steps :

Algorithm 20 Our dynamic-programming algorithm to solve the TCP problem

1: **for each** $b = w_i w_j \in B$ **do** ▷ B is the ordered set of nodes and consider that $i < j$
2: **while** δ_b is not empty **do**
3: $w_k \leftarrow \text{pop}(\delta_b)$;
4: **if** b is a leaf **then**
5: **if** $w_k = w_i$ **then**
6: Partition $[Q] \leftarrow \emptyset$; ▷ $Q = w_i w_j$
7: **else**
8: Partition $[Q] \leftarrow Q$; ▷ $Q = w_i w_j w_k$
9: **if** b is not a leaf **then**
10: **if** $w_k = w_i$ **then**
11: Partition $[Q] \leftarrow$ Partition $[(\text{leftChild}(b), w_i\rangle] \cup$ Partition $[(\text{rightChild}(b), w_i\rangle]$;
12: **else**
13: Partition $[Q] \leftarrow$ better of $\{$Partition $[(b, w_i\rangle] \cup w_i w_j w_k$, Partition $[(\text{leftChild}(b), w_k\rangle] \cup$ Partition $[(\text{rightChild}(b), w_k\rangle]\}$;
14: **return** Partition $[P_{1,2}] \cup$ Partition $[P_{2,1}]$;

1 - identify and organize the $2n$ cones into a multi-level DAG;

2 - build the stack δ_b for each cone b using Algorithm 19;

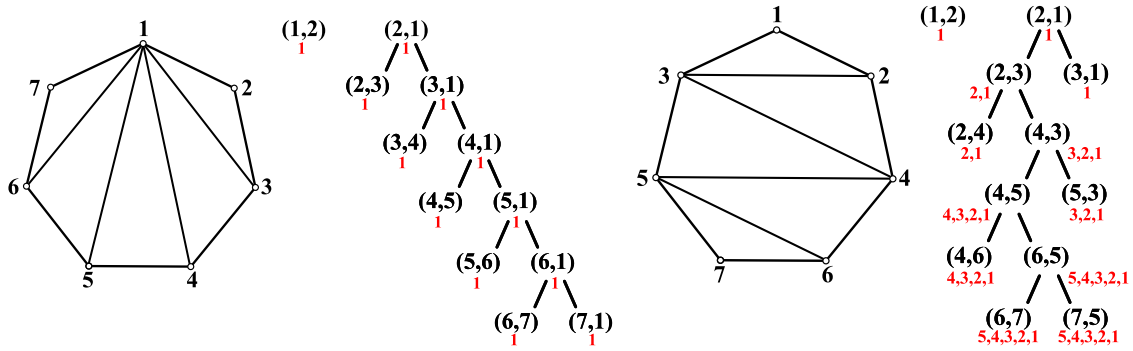
3 - evaluate these cones in a bottom-up manner as organized in step 1. This step is done by Algorithm 20.

Lemma 10 *Our dynamic-programming solution requires $O(n)$ execution time in some cases.*

Proof. In this solution, the DAG changes according to the input data and not as a consequence of the problem size. It makes it difficult to predict exactly how many operations are to be performed on each node. However, in some cases, each node will only have to compute a constant number $c \ll 2n$ of cones. For example in Figure 33a, each node will only have to execute one cone in the best case. ■

Theorem 9 *In the worst case, our dynamic-programming solution requires $O(n^2)$ execution time to solve the TCP problem.*

Proof. As there are at most $2n$ nodes in the DAG, it is deducible from Figure 33b that there are at most $(n-2)$ cones to compute for a given node in the worst case. ■



(a) *The best case* **(b)** *The worst case*

Figure 33 – *DAG in the best and the worst cases of a convex polygon P*

2.4.7 - Experimental results

Experimental setups

This section compares the proposed sequential solution with the previous best sequential and CGM-based parallel solutions. This experimentation has been performed on the Dolphin cluster of the MatriCS platform of the University of Picardie Jules Verne[5] using 48 nodes called thin nodes with 48 $\times$ 128GB of RAM, and 12 named thick nodes with 12 $\times$ 512GB of RAM. Each node is made of two Intel Xeon Processor E5-2680 V4 (35M Cache, 2.40 GHz), and each of them consists of 14 cores. All nodes are interconnected with OmniPath links providing 100Gbps throughput. These algorithms have been implemented in the C programming language[6], and on the operating system CentOS Linux release 7.6.1810. The MPI library (OpenMPI version 1.10.4) has been used for inter-processor communication. These algorithms have been executed on five thin nodes.

Note 1 *This experimental environment will also be used for the CGM-based parallel solutions described in Chapter 3.*

Five random data sets have been generated and tests have been carried out on each of them. The average of these results is presented according to different values of the pair (n, p). n is the data size, with values in the set $\langle$512, 1024, 2048, 4096, 8192, 12288, 16384, 20480, 24576, 28672, 32768, 36864, 40960, 45056, 49152, 53248, 57344, 61440, 65536$\rangle$, and p is the number of processors, with values in the set $\langle$1, 8, 16, 32$\rangle$. The proposed dynamic-programming solution is compared with the classical sequential algorithm of Yao (1982) and the improved sequential

5. https://www.u-picardie.fr/matrics
6. The source codes are available on this URL : https://github.com/compiii/Fast-Seq-Alg-for-MCOP.

algorithm by Myoupo and Kengne (2014a). This solution is also compared with the CGM-based parallel solution of Kechid and Myoupo (2008b) and Myoupo and Kengne (2014a). Table 2 gives an overview of the total execution time of these algorithms. For the CGM-based parallel solutions, it is the sum of the partitioning time, the computation time and the communication time. The CGM-based parallel solution proposed by Kechid and Myoupo (2008b) is named *Tcb*. Those of Myoupo and Kengne (2014a) are named *Tinit sort* and *Tmax sort*. The first one focuses on minimizing the number of communication rounds, and the second one on load balancing (Myoupo and Kengne, 2014a). Figures 34a, 34b, 34c, 34d, and 35 are drawn from results obtained in Table 2.

Evolution of the total execution time

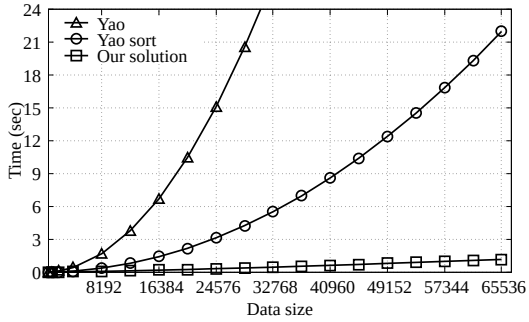
Figure 34a shows that the sequential solution of Myoupo and Kengne (2014a) significantly improves the total execution time compared to the sequential algorithm of Yao (1982). When $n = 65536$, the execution time has been reduced down to 79.55%. Nevertheless, by avoiding unnecessary computations, our solution reduced down to 94.72% the execution time and is $\times 18.93$ faster than the previous one. It is due to the drastic increase in the number of unnecessary computations when the data size increases. Figures 34b, 34c, and 34d show that the growth in number of processors does not outperform our solution. The execution time of *Tcb* decreases as the number of processors increases. One of the main causes is the use of the sequential algorithm of Yao (1982) in the local computation phases. Another reason is the high partitioning time. Indeed, each processor executes a partitioning algorithm requiring $O(n \times p)$ time. Although *Tinit sort* and *Tmax sort* reduce the complexity of this partitioning algorithm, it uses the sequential solution of Myoupo and Kengne (2014a) during the local computation phases.

Comparison of our solution and the best CGM-based parallel solution

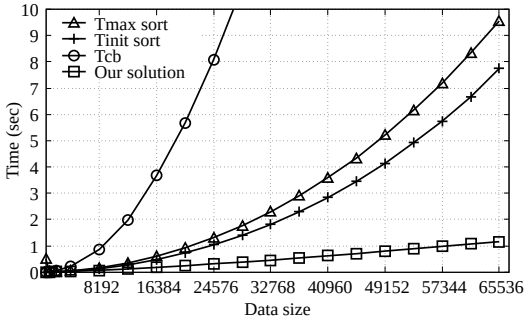
In Figure 35, we compare our solution with *Tinit sort*, which is the best CGM-based parallel solution, on large data sizes. Our solution narrows down the total execution time to 80.31% and is $\times 5.07$ faster than *Tinit sort* on thirty-two processors when $n = 65536$. It is easy to see the impact that the partitioning time has on the total execution time. When $n = 65536$, it is part of 52.67% on eight processors, 61.79% on sixteen processors, and 68.76% on thirty-two processors. Whatever the data size, it does not reduce as the number of processors increases. It is because this partitioning algorithm is not parallel. The ideal partitioning algorithm should be parallel with a complexity in $O(n/p)$ execution time and $O(1)$ communication round. Algorithms proposed to date scan the $2n$ nodes of the DAG to estimate the

Table 2 – *Total execution time (in seconds) of sequential and CGM-based parallel solutions while solving the TCP problem*

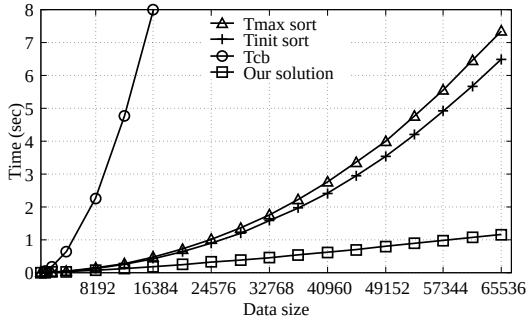
	Sequential solutions			CGM-based parallel solutions								
n	*Our sol.*	*Yao*	*Yao sort*	*Tinit sort*			*Tmax sort*			*Tcb*		
				$p=8$	$p=16$	$p=32$	$p=8$	$p=16$	$p=32$	$p=8$	$p=16$	$p=32$
512	0.001	0.01	0.001	0.001	0.004	0.008	0.009	0.02	0.02	0.006	0.03	0.06
1024	0.001	0.02	0.001	0.003	0.005	0.01	0.02	0.02	0.05	0.01	0.05	0.22
2048	0.01	0.11	0.02	0.01	0.01	0.03	0.01	0.01	0.01	0.06	0.18	0.72
4096	0.03	0.42	0.09	0.03	0.03	0.03	0.04	0.03	0.05	0.22	0.64	2.55
8192	0.07	1.68	0.37	0.12	0.12	0.15	0.16	0.14	0.14	0.86	2.25	9.02
12288	0.12	3.79	0.81	0.27	0.25	0.23	0.34	0.27	0.23	1.97	4.75	18.44
16384	0.18	6.68	1.41	0.47	0.42	0.39	0.60	0.47	0.42	3.66	7.99	28.85
20480	0.24	10.46	2.17	0.73	0.63	0.63	0.92	0.72	0.63	5.67	12.04	42.40
24576	0.32	15.07	3.13	1.03	0.89	0.85	1.31	1.01	0.87	8.08	16.76	57.81
28672	0.38	20.54	4.25	1.39	1.19	1.12	1.76	1.35	1.17	10.83	22.52	76.29
32768	0.45	26.67	5.52	1.81	1.58	1.43	2.29	1.75	1.51	14.05	29.22	98.24
36864	0.54	33.84	6.96	2.29	1.96	1.82	2.90	2.22	1.90	18.16	37.54	121.43
40960	0.62	41.90	8.60	2.83	2.41	2.20	3.59	2.75	2.34	22.64	46.13	148.44
45056	0.70	50.77	10.40	3.44	2.94	2.68	4.33	3.36	2.86	26.81	57.05	179.97
49152	0.80	60.54	12.39	4.13	3.53	3.26	5.21	3.98	3.42	34.26	70.90	221.40
53248	0.89	71.15	14.50	4.92	4.19	3.75	6.16	4.76	4.04	40.51	86.87	270
57344	0.98	82.62	16.82	5.73	4.91	4.43	7.18	5.56	4.76	49.32	108.69	336.44
61440	1.07	94.93	19.29	6.66	5.67	5.13	8.33	6.45	5.56	58.48	128.26	442.31
65536	1.16	107.40	21.96	7.74	6.49	5.89	9.55	7.34	6.31	63.90	157.93	494.87



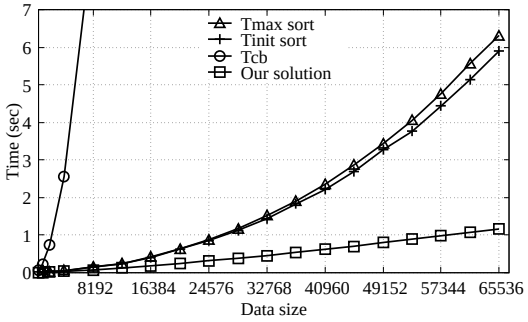
(a) *Our solution versus the sequential solutions*



(b) *Our solution versus the CGM-based parallel solutions on eight processors*



(c) *Our solution versus the CGM-based parallel solutions on sixteen processors*



(d) *Our solution versus the CGM-based parallel solutions on thirty-two processors*

Figure 34 – *Comparison of the total execution time between our solution versus the sequential solution of Yao (1982) and CGM-based parallel solutions of Kechid and Myoupo (2008b) and Myoupo and Kengne (2014a) while solving the TCP problem*

overall computational load to distribute tasks equitably onto processors. However, in many cases our solution requires $O(n)$ time. Thus, to parallelize our solution, the partitioning techniques of this DAG must be totally overhauled.

2.5 - Summary

In this chapter, we presented the work done on parallelization of sequential algorithms that solve the MPP, the MCOP, the TCP problem, and the OBST problem. On the CGM model, the proposed parallel solutions led to a trade-off between minimizing the number of communication rounds and balancing the load of processors. In (Kechid and Myoupo, 2008a, 2008b, 2009), the strategy consists in subdividing the DAG into small-size blocks to promote the load balancing. In (Kengne and Myoupo, 2012; Myoupo and Kengne, 2014a, 2014b), the strategy consists in subdividing the DAG into large-size blocks to minimize the number of communication rounds. Kengne et al. (2016) proposed a CGM-based parallel solution that gives the

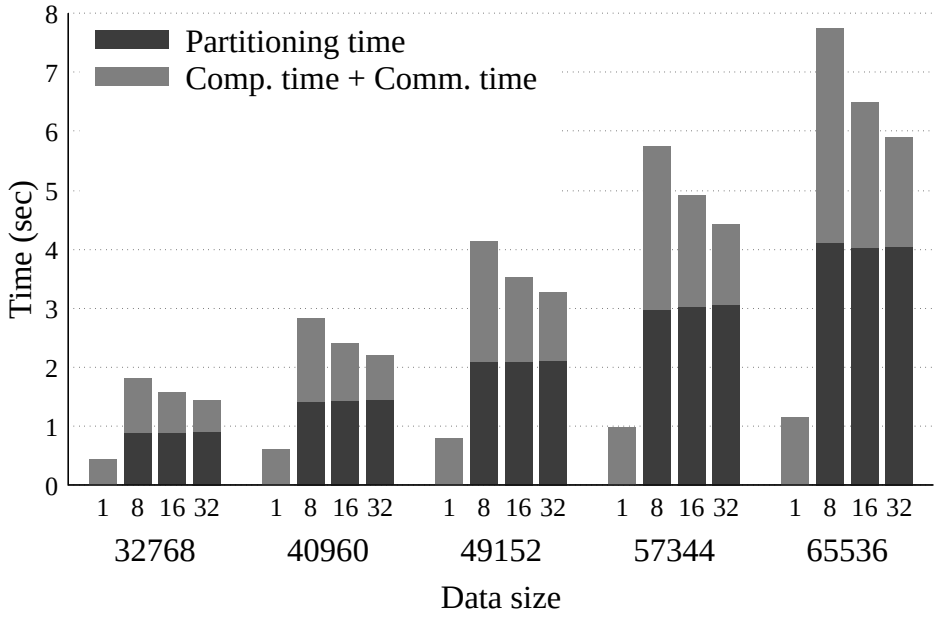


Figure 35 – *Comparison of our solution and the best CGM-based parallel solution while solving the TCP problem*

end-user the choice to optimize one criterion according to their own goal. However, the common problem related to these criteria, regardless of the end-user's choice, is the high global communication time that is caused by the high latency time of processors.

For the MCOP and the TCP problem, we proposed a fast sequential algorithm which, while not solving this trade-off, is faster than previous sequential and CGM-based parallel algorithms (Kechid and Myoupo, 2008b; Myoupo and Kengne, 2014a; Yao, 1982). We noticed that these solutions did not take into account the dependencies of cones during their evaluations. They evaluated cones according to their order of precedence. We have constructed for each node of the DAG a stack of cones on which it depends, through an algorithm that requires $O(n)$ execution time in the worst case, and we have evaluated each node from this stack. Experimental results performed on the MatriCS platform showed that our sequential algorithm is $\times 18.93$ faster than the sequential algorithm of Yao (1982) and $\times 5.07$ faster than the CGM-based parallel solution of Myoupo and Kengne (2014a) on thirty-two processors. Chapter 3 will outline CGM-based parallel solutions that reconcile these conflicting objectives to solve the MPP and the OBST problem.

CHAPTER 3

Reconciliation of the Minimization of the Number of Communication Rounds and the Load-Balancing of Processors

CONTENTS

3.1 - Introduction

This chapter begins by designing, in Section 3.2, a dynamic graph model to solve the OBST problem and showing that each instance of this problem corresponds to a one-to-all shortest path problem in this dynamic graph (like in (Bradford, 1994) to solve the MPP). Thereafter, it describes our CGM-based parallel solutions that address the trade-off of minimizing the number of communication rounds and balancing the load of processors. In computation rounds, the sequential algorithm of Godbole (1973) is used to solve the MPP and the sequential algorithm of Knuth (1971) is used to solve the OBST problem. These solutions are respectively based on three dynamic graph partitioning techniques presented in Sections 3.3, 3.4, and 3.5 : the irregular partitioning technique, the k-block splitting technique, and the four-splitting technique. In these sections, experimental results are presented to show the performance of each technique.

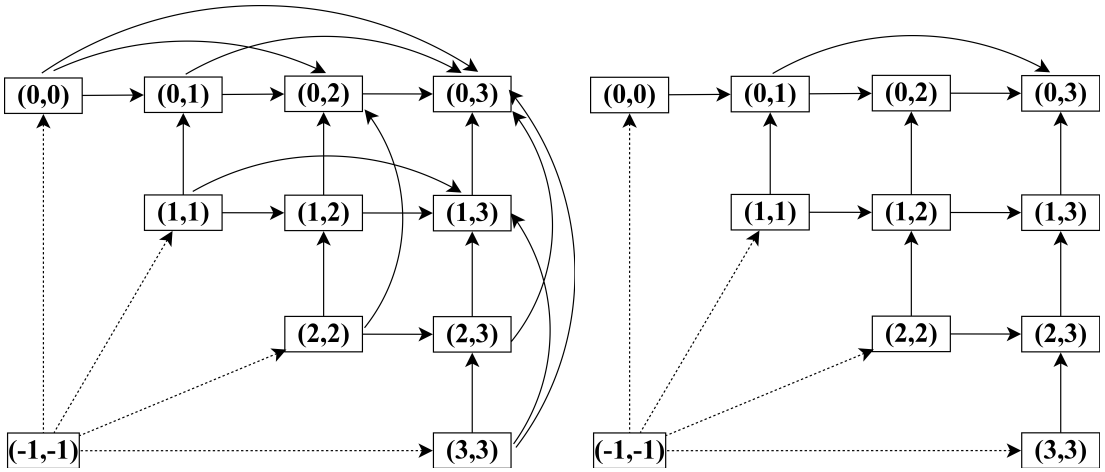
3.2 - Dynamic graph model of the OBST problem

Denote by $K = \langle k_1, k_2, \ldots, k_n \rangle$ a set of n sorted keys, such that $k_1 < k_2 < \cdots < k_n$. Some searches may concern values that do not belong to K; so consider a set of $(n+1)$ dummy keys $D = \langle d_0, d_1, \ldots, d_n \rangle$ representing values outside K. For each key k_i (respectively dummy key d_i), p_i (respectively q_i) is the probability that a search concerns k_i (respectively d_i). The goal is to build an optimal binary search tree from these keys. Each key k_i is an internal node and each dummy key d_i is a leaf. The DP formalization of the OBST problem has been carried out in Section 2.3.1. Let us define the dynamic graph D_n of the OBST problem by a set of vertices (i, j) and a set of edges, such that $0 \leq i \leq j \leq n$.

Two vertices (i, j) and (k, m) are on the same row (respectively on the same column) if $i = k$ (respectively $j = m$). They are on the same diagonal $(j - i + 1)$ if $(j - i) = (m - k)$. D_n consists of $(n+1)$ diagonals of vertices, $(n+1)$ rows of vertices, and $(n+1)$ columns of vertices. The cost of the shortest path from an added virtual vertex $(-1, -1)$ to any vertex (i, j) is stored in $SP[i, j]$, where SP is a shortest path matrix of size $(n+1)$.

The unit edges are denoted by $\rightarrow$, $\uparrow$, and $\nearrow$. The unit edge $(i, j) \rightarrow (i, j+1)$ has a weight equal to $q_{j+1} + w(i, j+1)$ and represents the associative product $(k_{i+1} \bullet \cdots \bullet k_j) \bullet k_{j+1}$. It is a binary subtree of root k_{j+1} whose the left part contains the set of keys $K_{i,j} = \langle k_{i+1}, k_{i+2}, \ldots, k_j \rangle$ and the set of dummy keys $D_{i,j} = \langle d_i, d_{i+1}, \ldots, d_j \rangle$, and the right part contains the dummy key d_{j+1}. Similarly, the unit edge $(i, j) \uparrow (i-1, j)$ has a weight equal to $q_{i-1} + w(i-1, j)$ and represents the product $k_i \bullet (k_{i+1} \bullet \cdots \bullet k_j)$, corresponding to a binary subtree of root k_i whose the left part contains the dummy key d_{i-1}, and the right part contains the set of keys $K_{i,j-1} = \langle k_{i+1}, k_{i+2}, \ldots, k_{j-1} \rangle$ and the set of dummy keys $D_{i,j} = \langle d_i, d_{i+1}, \ldots, d_j \rangle$. For all i such that $0 \leq i \leq n$, the arrows $\nearrow$ represent unit edges from $(-1, -1)$ to (i, i) and their weights are equal to q_i.

To represent the split tree, that is, a tree splits into two subtrees, D_n is made up of edges called *jumps*. Denote a horizontal jump by $\Rightarrow$, and a vertical jump by $\Uparrow$. The vertical jump $(i, j) \Uparrow (s, j)$ is $(i - s)$ units long and the horizontal jump $(i, j) \Rightarrow (i, t)$ is $(t - j)$ units long. All non-jump edges are unit edges of length 1. The shortest path to the node (i, j) through the jumps $(i, a) \Rightarrow (i, j)$ and $(a+1, j) \Uparrow (i, j)$, represented by the product $(k_{i+1} \bullet \cdots \bullet k_a) \bullet (k_{a+1} \bullet \cdots \bullet k_j)$, define the same binary subtree of root k_{a+1}. The left part of this subtree contains the set of keys $K_{i,a} = \langle k_{i+1}, k_{i+2}, \ldots, k_a \rangle$ and the set of dummy keys $D_{i,a} = \langle d_i, d_{i+1}, \ldots, d_a \rangle$, and the right part contains the set of keys $K_{a+1,j} = \langle k_{a+2}, k_{a+3}, \ldots, k_j \rangle$ and the set of



(a) *Dynamic graph* D_3 **(b)** *Dynamic graph* D'_3

Figure 36 – *Dynamic graphs* D_3 *and* D'_3 *for a problem of size* $n = 3$

dummy keys $D_{a+1,j} = \langle d_{a+1}, d_{a+2}, \ldots, d_j \rangle$. Definition 5 gives the formal definition of D_n and Figure 36a depicts D_3, which refers to a problem of size $n = 3$.

Definition 5 *Given a set of n sorted keys, a dynamic graph* $D_n = (V, E \cup E')$ *is defined as a set of vertices,*

$$V = \{(i,j) : 0 \leq i \leq j \leq n\} \cup \{(-1,-1)\}$$

a set of unit edges,

$$E = \{(i,j) \to (i,j+1) : 0 \leq i \leq j < n\} \cup \{(i,j) \uparrow (i-1,j) : \\ 0 < i \leq j \leq n\} \cup \{(-1,-1) \nearrow (i,i) : 0 \leq i \leq n\}$$

a set of jumps,

$$E' = \{(i,j) \Rightarrow (i,t) : 0 \leq i < j < t \leq n\} \cup \{(s,t) \Uparrow (i,t) : \\ 0 \leq i < s < t \leq n\}$$

a weight function W such that

$$\begin{aligned}
&W((i,j) \to (i,j+1)) = q_{j+1} + w(i,j+1) && 0 \leq i \leq j < n \\
&W((i,j) \uparrow (i-1,j)) = q_{i-1} + w(i-1,j) && 0 < i \leq j \leq n \\
&W((-1,-1) \nearrow (i,i)) = q_i && 0 \leq i \leq n \\
&W((i,k) \Rightarrow (i,j)) = SP[k+1,j] + w(i,j) && 0 \leq i < k < j \leq n \\
&W((k+1,j) \Uparrow (i,j)) = SP[i,k] + w(i,j) && 0 \leq i \leq k < j \leq n
\end{aligned}$$

There exists a similarity between the shortest path matrix SP and the DP table $Tree$ (see Equation (2.4)) because the value of $SP[i,j]$ in the dynamic graph D_n is

identical to $Tree[i,j]$ in the DP table. Computing a shortest path from $(-1,-1)$ to (i,j) gives the minimum cost of finding the OBST represented by the product $k_{i+1} \bullet k_i \bullet \cdots \bullet k_j$, where $0 \leq i < j \leq n$. Therefore, finding the shortest path from (i,j) to $(0,n)$ gives the minimum cost of finding the OBST represented by the product $k_1 \bullet \cdots \bullet k_i \bullet P \bullet k_{j+1} \bullet \cdots \bullet k_n$, where $P = (k_{i+1} \bullet \cdots \bullet k_j)$.

Lemma 11 *For all vertices (i,k) in D_n, $SP[i,k]$ can be computed by a path having edges of length no larger than $\lceil (k-i)/2 \rceil$.*

Proof. Suppose that $(i,j) \Rightarrow (i,k)$ is in a shortest path to (i,k) and $k-j > \lceil (k-i)/2 \rceil$. Hence, $SP[i,k] = SP[i,j] + W((i,j) \Rightarrow (i,k)) = SP[i,j] + SP[j+1,k] + w(i,k)$. But $W((j+1,k) \Uparrow (i,k)) = SP[i,j] + w(i,k)$, so $SP[i,k] = SP[j+1,k] + W((j+1,k) \Uparrow (i,k))$. The jump $(j+1,k) \Uparrow (i,k)$ is of length $j+1-i$. Therefore, since $j+1-i+k-j = k-i+1$ and $k-j > \lceil (k-i)/2 \rceil$, it can be deduced that $j+1-i \leq \lceil (k-i)/2 \rceil$. On the other hand, a shortest path to $(j+1,k)$ cannot contain a jump longer than $k-(j+1)$. Since $k-(j+1) < k-j$, this lemma follows inductively. ■

The proof of Lemma 11 leads directly to Theorem 10.

Theorem 10 (Duality Theorem) *If a shortest path from $(-1,-1)$ to (i,j) contains the jump $(i,k) \Rightarrow (i,j)$, then there is a dual shortest path containing the unit edge $(k+1,j) \uparrow (i,j)$ when $0 \leq i = k < j \leq n$ and the jump $(k+1,j) \Uparrow (i,j)$ when $0 \leq i < k < j \leq n$.*

Theorem 10 is fundamental because it makes possible to avoid redundant computations when looking for the value of the shortest path of D_n's vertices. Indeed, for any vertex (i,j), among all its shortest paths containing jumps, only those that contain only horizontal jumps are evaluated. So, the input graph of our CGM-based parallel solutions is a subgraph of D_n denoted by D'_n, in which the set of edges from (i,k) to (i,j), such that $0 \leq i = k < j \leq n$, and from $(k+1,j)$ to (i,j), such that $0 \leq i < k < j \leq n$, is removed. Figure 36b shows the dynamic graph D'_3.

From Figures 13b and 36b, a difference between our dynamic graph D'_n and the one proposed by Bradford (1994) to solve the MPP (described in Section 2.2.3) can be noticed. Indeed, Bradford (1994) only removed vertical jumps in the dynamic graph D'_n. However, when $0 \leq i = k < j \leq n$, the shortest path going from (i,k) to (i,j), corresponding to the horizontal jump $(i,k) \Rightarrow (i,j)$, has the same value as the one from $(k+1,j)$ to (i,j), corresponding to the unit edge $(k+1,j) \uparrow (i,j)$. This is the reason why in addition to removing vertical jumps, these horizontal jumps in D'_n have also been removed to avoid more redundant computations.

Theorem 11 *Each instance of an OBST problem of size n corresponds to a dynamic graph D_n, where $SP[i,j]$ is equal to $Tree[i,j]$. As a result, the shortest path from the virtual vertex $(-1,-1)$ to the vertex $(0,n)$ in D_n solves $Tree[0,n]$.*

Proof. To prove this theorem, it is sufficient to show that the cost of every path from $(-1,-1)$ to $(0,n)$ in D_n corresponds to the cost of construction of an OBST of n elements associative product. In addition, for every tree of n elements associative product, there is a corresponding path in D_n where both the path and the associative product have the same cost.

We only show that for each path from $(-1,-1)$ to $(0,n)$ in D_n, there is a corresponding associative product of n elements (or $n-1$ operators "$\bullet$-s"). This proof is by induction on lengths of associative products. Consider D_4 and the product $k_1 \bullet k_2 \bullet k_3 \bullet k_4$. In D_4, the jump $(0,2) \Rightarrow (0,4)$ has weight $SP[3,4]+w(0,4)$. Therefore, the cost of a path using this jump corresponds to the cost of the associative product $(k_1 \bullet k_2) \bullet (k_3 \bullet k_4)$. A symmetric argument holds for the jump $(3,4) \Uparrow (0,4)$. Hence for each associative product there is a path, and for each path there is an associative product.

Now suppose that the theorem holds for all $n \leq k$, where $k \geq 4$ and n is the number of associative operators in a given associative product. Thus, the inductive hypothesis is for any path in D_n from $(-1,-1)$ to $(0,n)$, where $n \leq k$, there is a corresponding associative product of n elements with the same cost. Without loss of generality, consider horizontal jumps. For $m = 2k$, let's take a path in D_m from $(-1,-1)$ to $(0,m)$. In the following, it will show that for all $m \leq 2k$, there is a corresponding m elements associative product of the same cost (this proof holds for both even and odd length products because D_{k-1} is a proper subgraph of D_k).

The structure of D_m along with the inductive hypothesis gives:

1 - The cost of a path from $(-1,-1)$ to any node (i,j), such that $j-i \leq k$, corresponds to the cost of a binary search tree of $k_{i+1} \bullet \cdots \bullet k_j$ by the inductive hypothesis. Since the product $k_{i+1} \bullet \cdots \bullet k_j$ contains at most $k-1$ associative operators ("$\bullet$-s").

2 - The cost of a path from (i,t) to $(1,2k) = (0,m)$, $(1,2k) = (0,m)$, such that $k \leq t-i < m$, corresponds to the cost of a binary search tree of $k_1 \bullet \cdots \bullet k_i \bullet P \bullet k_{t+1} \bullet \cdots \bullet k_m$, where $P = (k_{i+1} \bullet \cdots \bullet k_t)$, by the inductive hypothesis. Since P consists of at least k elements, the product $k_1 \bullet \cdots \bullet k_i \bullet P \bullet k_{t+1} \bullet \cdots \bullet k_m$ consists of at most $(k-1)$ associative operators ("$\bullet$-s").

Suppose that path from $(-1,-1)$ to $(1,2k)$ includes the jump $(i,j) \Rightarrow (i,t)$. Then assume that $j-i < k \leq t-i$ (otherwise by the inductive hypothesis and the two facts

above, the proof is complete). Therefore, consider the jump $(i,j) \Rightarrow (i,t)$, where $j-i < k \leq t-i$. By the inductive hypothesis and the fact 2, the cost of a path from (i,t) to $(0,m)$ corresponds to the cost of a binary search tree of $k_1 \bullet \cdots \bullet k_i \bullet P \bullet k_{t+1} \bullet \cdots \bullet k_m$, where $P = (k_{i+1} \bullet \cdots \bullet k_t)$. Again by the inductive hypothesis and the fact 1 above, the cost of a path to (i,j) corresponds to the cost of a binary search tree of $k_{i+1} \bullet \cdots \bullet k_j$, where $k_i \bullet \cdots \bullet k_j$ is a subproduct of P. Furthermore, the jump $(i,j) \Rightarrow (i,t)$ is a part of a path from $(-1,-1)$ to $(0,m)$ whose cost corresponds to the cost of the product $(k_{i+1} \bullet \cdots \bullet k_j) \bullet (k_{j+1} \bullet \cdots \bullet k_t)$. This is because its cost is $SP[j+1,t] + w(i,t)$ and because $t-(j+i) < j-i$, the inductive hypothesis can be applied. Vertical jumps follow similarly. ■

Corollary 1 *Any solution of a one-to-all shortest path problem can be used on our dynamic graph D'_n to solve the OBST problem.*

Figure 37 shows the dynamic graph D'_3 that was filled from a set of three sorted keys with probabilities given in Table 1. The weights of edges were computed using the shortest path matrix *SP* (which is similar to the DP table *Tree*) and starting with the first diagonal containing the vertices $(0,0)$, $(1,1)$, $(2,2)$, and $(3,3)$, then the second diagonal containing the vertices $(0,1)$, $(1,2)$, and $(2,3)$, then the third diagonal containing the vertices $(0,2)$ and $(1,3)$, and finally the last diagonal containing the vertex $(0,3)$. The shortest path from $(-1,-1)$ to $(0,3)$ is colored in red in Figure 37. It is equal to $0.03+0.4+1.37 = 1.8$ (note that it is the same value as $Tree[0,3]$ in Figure 23a). The optimal binary search tree corresponding to this path is also depicted. Now let's outline our first dynamic graph partitioning.

3.3 - First dynamic graph partitioning : irregular partitioning technique

The irregular partitioning technique consists in subdividing the shortest path matrix into submatrices (blocks) of varying size (irregular size) to allow a maximum of processors to remain active longer. The idea is to increase the number of blocks of diagonals whose this number is lower or equal to half of the first one through the block fragmentation technique. This technique aims to reduce the block size by dividing it into four subblocks. Figure 38 shows the subblock sizes obtained by fragmenting a block of arbitrary size $\alpha \times \beta$.

To minimize the number of communication rounds, it begins to subdivide the shortest path matrix with large-size blocks from the largest diagonal (the first diag-

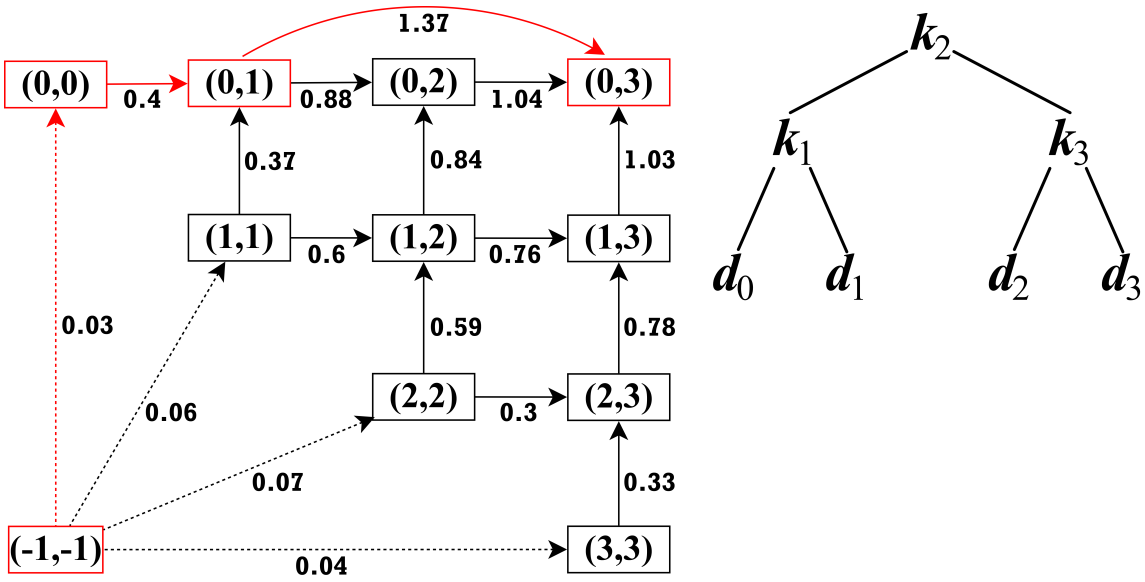


Figure 37 – *Dynamic graph D'_3 filled while determining the optimal binary search tree from probabilities of three sorted keys given in Table 1*

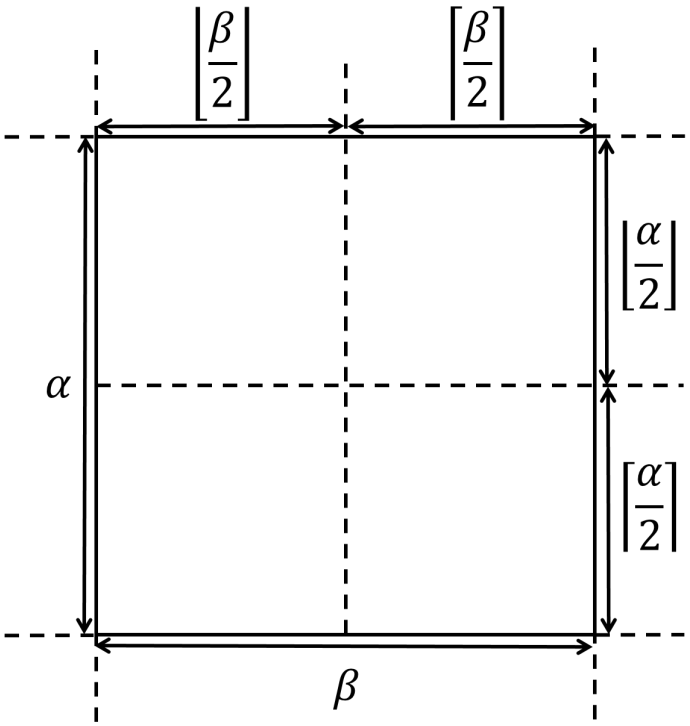


Figure 38 – *Fragmentation of a block of size* $\alpha \times \beta$

onal of blocks) to the diagonal located just before the one whose number of blocks is half of the first one. Then, since the number of blocks per diagonal quickly becomes smaller than the number of processors, to increase the number of blocks of these diagonals and allow a maximum of processors to remain active, it fragments all the blocks belonging to the next diagonal until the last one to catch up or exceed by one notch the number of blocks of the first diagonal. It reduces the idle time

of processors and promotes the load balancing. This process is repeated k time, after which the block sizes are no longer modify, and the rest of the partitioning becomes traditional because an excessive fragmentation would lead to a drastic rise of the number of communication rounds. After performing k fragmentations, a block belonging to the lth level of fragmentation have been subdivided l times, $0 \leq l \leq k$.

In the following, let us consider that the shortest path matrix is of size n for both the MPP and the OBST problem. Formally, denoting by $f(p) = \lceil \sqrt{2p} \rceil$, $\theta(n,p) = \left\lceil \frac{n}{f(p)} \right\rceil$, and $\theta(n,p,l) = \left\lceil \frac{\theta(n,p)}{2^l} \right\rceil$, we subdivide the shortest path matrix SP into blocks denoted by $SM(i,j)$. $SM(i,j)$ is a matrix of size $\theta(n,p,l) \times \theta(n,p,l)$ belonging to the lth level of fragmentation. Equation (3.1) shows entries of SP delimiting a block $SM(i,j)$, such that $1 \leq i \leq j \leq n$:

$$SM(i,j) = \begin{pmatrix} SP[i, j-\theta(n,p,l)+1] & \cdots & SP[i,j] \\ \vdots & \cdots & \vdots \\ SP[i+\theta(n,p,l)-1, j-\theta(n,p,l)+1] & \cdots & SP[i+\theta(n,p,l)-1, j] \end{pmatrix} \tag{3.1}$$

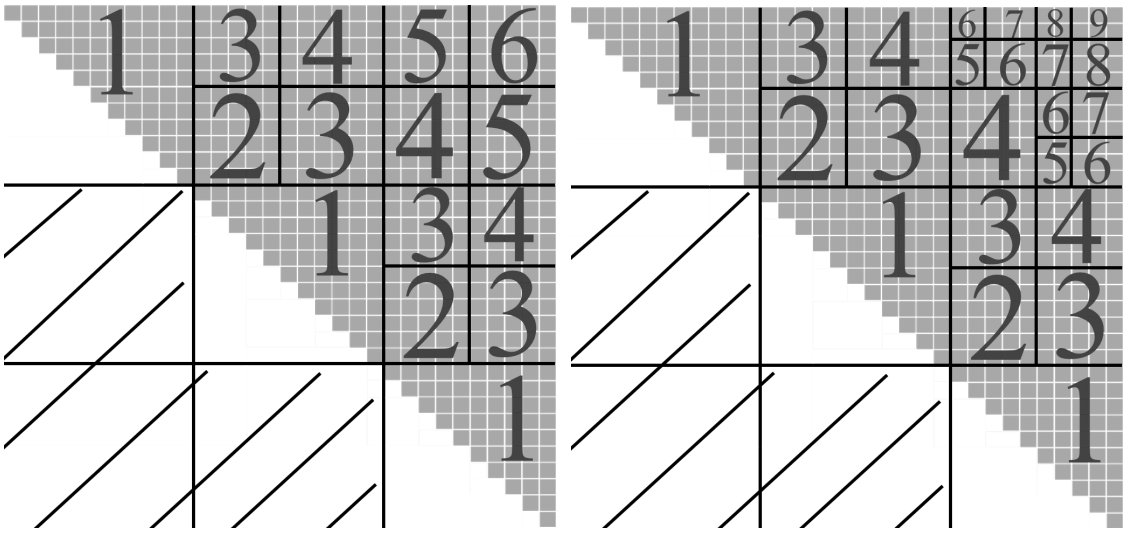
Figures 39a, 39b, 39c, and 39d illustrate four scenarios of this partitioning for $n = 32$, $k \in \{1,2\}$, and $p \in \{3, 4, 5, 6, 7, 8\}$. The number in each block represents the diagonal in which it belongs. In Figure 39d, there are four diagonals of blocks when no fragmentation is performed (like in Figure 18c). Since the number of blocks of the third diagonal is equal to the half of the first, the first fragmentation is performed on the block of this diagonal (the blocks of size 8×8 is divided in four blocks of size 4×4) and the number of diagonals of blocks increases up to three (see Figure 39c). Then, since the number of blocks of the sixth diagonal is equal to the half of the first, the second fragmentation is performed on the blocks of this diagonal (the blocks of size 4×4 is divided in four blocks of size 2×2) and the number of diagonals of blocks increases up to three. Finally, the shortest path matrix contains ten diagonals of blocks.

Remark 1 *Some relevant points can be noticed about this partitioning :*

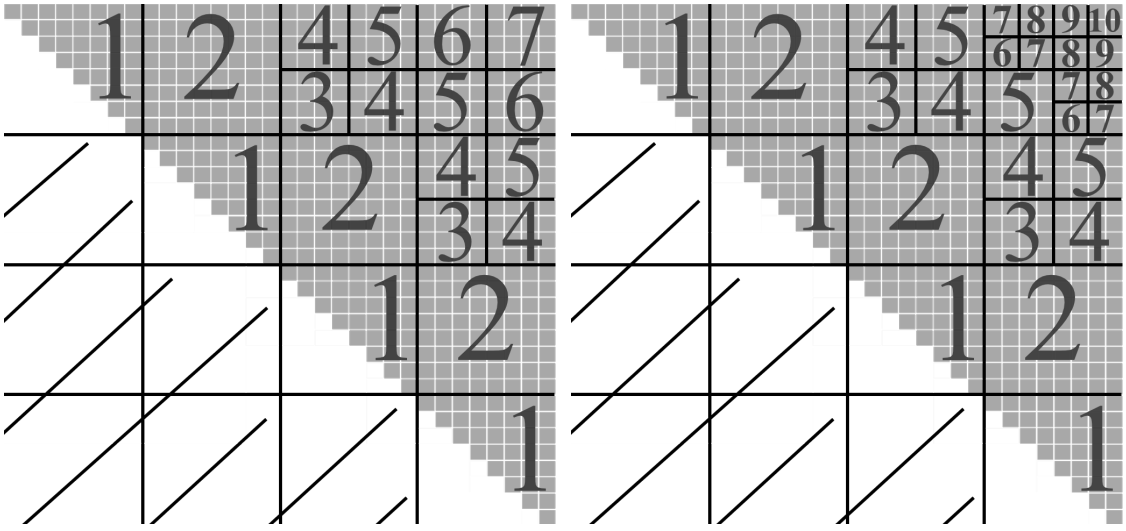
1 - *the blocks of the first diagonal are upper triangular matrices of* $\theta(n,p)$ *rows and* $\theta(n,p)$ *columns;*

2 - *all the blocks are usually not full when* $n \bmod \left(2^k \times f(p)\right) \neq 0$*; for example in Figure 39b where* $32 \bmod \left(2^2 \times 3\right) = 8 \neq 0$*;*

3 - *a block belonging to the* l*th level of fragmentation is full if it is a non-triangular matrix of size* $\theta(n,p,l) \times \theta(n,p,l)$*;*



(a) $p \in \{3, 4\}$ *and* $k = 1$ **(b)** $p \in \{3, 4\}$ *and* $k = 2$



(c) $p \in \{5, 6, 7, 8\}$ *and* $k = 1$ **(d)** $p \in \{5, 6, 7, 8\}$ *and* $k = 2$

Figure 39 – *Irregular partitioning technique of the shortest path matrix for* $n = 32$, $k \in \{1, 2\}$, *and* $p \in \{3, 4, 5, 6, 7, 8\}$. *For* $p \in \{3, 4\}$, *SP is partitioned into fifteen blocks when* $k = 1$ *and into twenty-four blocks when* $k = 2$. *For* $p \in \{5, 6, 7, 8\}$, *SP is partitioned into nineteen blocks when* $k = 1$ *and into twenty-eight blocks when* $k = 2$

4 - one fragmentation increases up to $(\lceil f(p)/2 \rceil + 1)$ *the number of diagonals (see proof of Lemma 13);*

5 - when $f(p)$ *is odd, the number of blocks in a diagonal after each fragmentation exceeds by one notch the number of blocks of the first diagonal. This is illustrated in Figure 39a, where there are three blocks in the first diagonal and four blocks in third diagonal.*

Lemma 12 *The number of blocks of the shortest path matrix after partitioning is a function of k (the number of fragmentations) and is equal to:*

$$C = \frac{S(S+1)}{2} + \frac{k}{2}\left[(S+1)(S+2\Delta) - \left\lceil\frac{S}{2}\right\rceil\left(\left\lceil\frac{S}{2}\right\rceil - 1\right)\right]$$

with $S = f(p)$ *and* $\Delta = (S \bmod 2)$.

Proof. After partitioning, there is exactly $S(S+1)/2 - \lceil S/2\rceil\ (\lceil S/2\rceil + 1)/2$ larger-size blocks. Depending on the parity of S, two scenarios can arise :

1 - When S is even, there is $(k-1) \times (S(S+1)/2 - \lceil S/2\rceil\ (\lceil S/2\rceil - 1)/2)$ blocks in diagonals from the first to the $(k-1)$th fragmentation (for example the second, the third, and the fourth diagonal on Figure 39b). This number increases up to $(S(S+1)/2 + \lceil S/2\rceil)$ blocks after the kth fragmentation.

2 - When S is odd, the principle is the same as in the previous scenario, except that here the fragmentation increases up to $(S+1)$ additional blocks on the initial block numbers (see point 5 of Remark 1).

Denote by $\Delta = (S \bmod 2)$ the variable, which determines the parity of S. Thus, we have:

$$\begin{aligned}
C &= (k-1) \times \left(\frac{S(S+1) - \left\lceil\frac{S}{2}\right\rceil\left(\left\lceil\frac{S}{2}\right\rceil - 1\right)}{2} + \Delta(S+1)\right) + \left\lceil\frac{S}{2}\right\rceil + \frac{S(S+1)}{2} \\
&\quad + \Delta(S+1) + \frac{S(S+1) - \left\lceil\frac{S}{2}\right\rceil\left(\left\lceil\frac{S}{2}\right\rceil + 1\right)}{2} \\
&= (k-1) \times \frac{(S+1)(S+2\Delta) - \left\lceil\frac{S}{2}\right\rceil\left(\left\lceil\frac{S}{2}\right\rceil - 1\right)}{2} + (S+1)(S+\Delta) + \left\lceil\frac{S}{2}\right\rceil\frac{1 - \left\lceil\frac{S}{2}\right\rceil}{2} \\
&= \frac{S(S+1)}{2} + \frac{k}{2}\left[(S+1)(S+2\Delta) - \left\lceil\frac{S}{2}\right\rceil\left(\left\lceil\frac{S}{2}\right\rceil - 1\right)\right]
\end{aligned}$$

■

Lemma 13 *Our irregular partitioning technique of the dynamic graph induces* $f(p) + k \times (\lceil f(p)/2\rceil + 1)$ *diagonal of blocks when the blocks undergo k successive fragmentations.*

Proof. If no fragmentation is performed (if $k = 0$), then there are $f(p)$ diagonals. Suppose there are k fragmentations. One fragmentation increases up to $(\lceil f(p)/2\rceil + 1)$ the number of diagonals. Indeed, a fragmentation is performed

when the number of blocks of a diagonal is equal to $\lceil f(p)/2 \rceil$. This fragmentation, which decreases the size of blocks by 1/4, creates a diagonal of $\lceil f(p)/2 \rceil$ blocks; then the following diagonals contain successively $2 \times \lceil f(p)/2 \rceil$ blocks, $(2 \times \lceil f(p)/2 \rceil - 1)$ blocks,..., $(\lceil f(p)/2 \rceil + 1)$ blocks. This is equal to $(\lceil f(p)/2 \rceil + 1)$ diagonals (for example the second, the third, and the fourth diagonal on Figure 39b). At the next diagonal, another fragmentation is done. We conclude from all this that after k fragmentations, we have $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ diagonals. ■

3.3.1 - Blocks' dependency analysis

Blocks' dependency of the MPP

Dependencies between blocks is given by Theorem 12, which is derived from Lemmas 1 and 2. It shows that blocks on the same diagonal are independent.

Theorem 12 (Blocks' dependency) *To evaluate the shortest paths of nodes of the block $SM(i,j)$ belonging to the lth level of fragmentation, the shortest path values to every node in blocks $SM(i, j - \theta(n,p,l))$, $SM(i, j - 2 \times \theta(n,p,l))$,...,$SM(i, j - (u-1) \times \theta(n,p,l))$, and $SM(i + \theta(n,p,l), j)$, $SM(i + 2 \times \theta(n,p,l), j)$,...,$SM(i + (u-1) \times \theta(n,p,l), j)$ are required, where $u = \lceil (j-i)/\theta(n,p,l) \rceil$.*

Figure 40 illustrates an example of dependencies of two blocks $SM(i,j)$ and $SM(h,l)$ with other blocks (the most shaded blocks). Evaluating the shortest paths of nodes of blocks belonging to the same diagonal can be carried out in parallel.

Blocks' dependency of the OBST problem

Figures 41a and 41b respectively show an example of dependencies and extremities of a block $SM(i,j)$ belonging to the lth level of fragmentation after applying the irregular partitioning technique. The extremities of $SM(i,j)$ are defined by :

- the leftmost upper entry $LUE_{ij} = (i, j - \theta(n,p,l) + 1)$;
- the rightmost upper entry $RUE_{ij} = (i,j)$;
- the leftmost lower entry $LLE_{ij} = (i + \theta(n,p,l) - 1, j - \theta(n,p,l) + 1)$;
- the rightmost lower entry $RLE_{ij} = (i + \theta(n,p,l) - 1, j)$.

Figure 41a shows eight points (A, B, C, D, E, F, G, and H) that identify blocks on which the block $SM(i,j)$ depends :

- $A = Tree\left[LUE_{i,j-3\times\theta(n,p,l)-1}\right]$

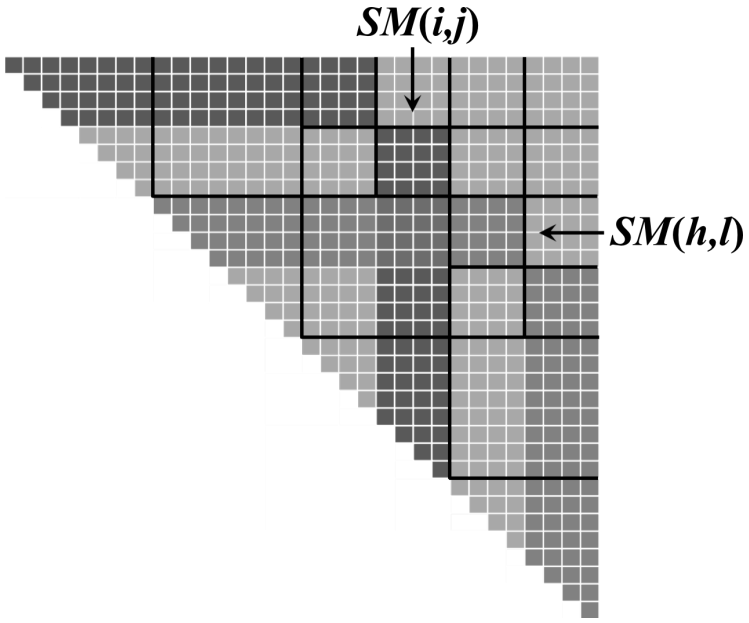
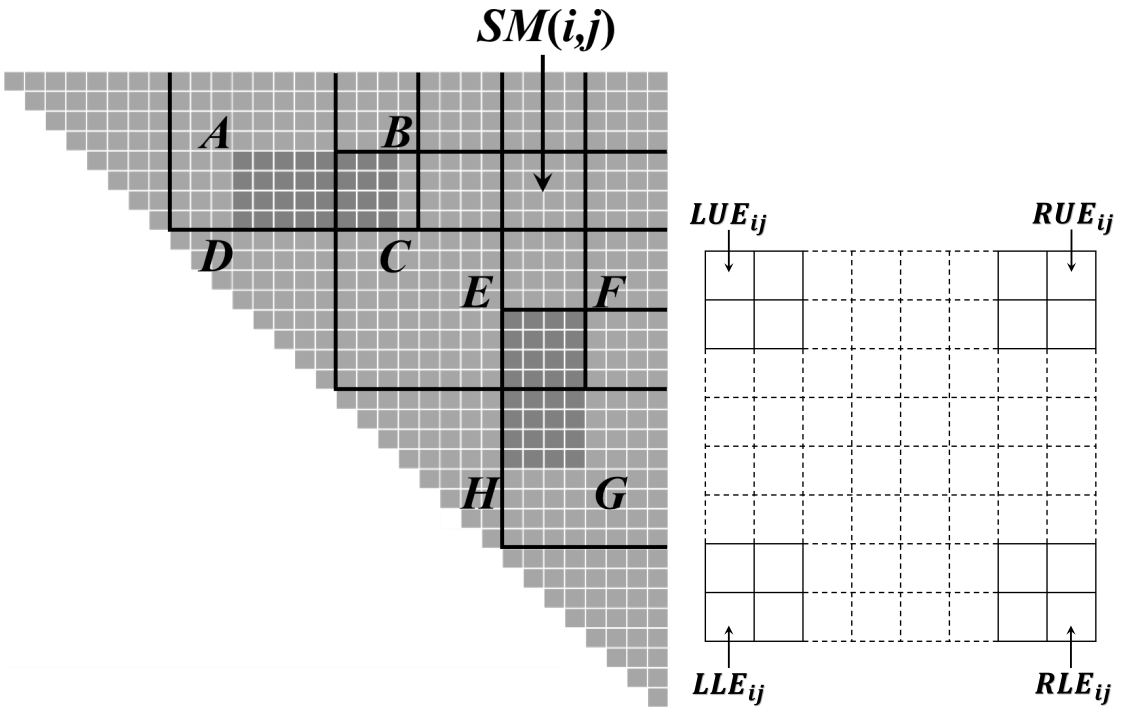


Figure 40 – *Dependencies of two blocks $SM(i, j)$ and $SM(h,l)$ after applying the irregular partitioning technique*

- $B = Tree\left[LUE_{i,j-\theta(n,p,l)-2}\right]$
- $C = Tree\left[LLE_{i,j-\theta(n,p,l)-2}\right]$
- $D = Tree\left[LLE_{i,j-3\times\theta(n,p,l)-1}\right]$
- $E = Tree\left[LLE_{i+\theta(n,p,l)+1,j}\right]$
- $F = Tree\left[RLE_{i+\theta(n,p,l)+1,j}\right]$
- $G = Tree\left[RLE_{i+3\times\theta(n,p,l),j}\right]$
- $H = Tree\left[LLE_{i+3\times\theta(n,p,l),j}\right]$

As in (Kengne et al., 2016), all lower blocks and subblocks (a part of a block) that are in the same row and column as the block $SM(i, j)$ are no longer absolutely required to evaluate $SM(i, j)$ because of the speedup of Knuth (1971), which does not allow estimating the exact load of a block before its evaluation (Kengne, 2014). The blocks on which the block $SM(i, j)$ depends can be carried out in parallel because they are not located on the same diagonal as $SM(i, j)$ and those belonging to the same diagonal are independent.

Remark 2 *To minimize the amount of data exchanged between processors in communication rounds, each processor must communicate only the subblock of its block, corresponding to the size of the target block.*



(a) *Dependencies of* $SM(i,j)$ **(b)** *Extremities of* $SM(i,j)$

Figure 41 – *Dependencies and extremities of a block* $SM(i,j)$ *after applying the irregular partitioning technique*

3.3.2 - Mapping blocks onto processors

We use a snake-like mapping scheme introduced in (Kengne and Myoupo, 2012) to allow some processors to evaluate at most one block more than the others. This scheme consists of assigning all blocks of a given diagonal from the leftmost upper corner to the rightmost lower corner. This process is reiterated until a block has been assigned to each processor, starting from processor P_0 and traveling through the blocks with a snake-like path. Figure 42 illustrates this mapping on six processors.

This mapping allows the processors to remain active as soon as possible. It also ensures the load balancing because it distributes equally small and large size blocks among processors. However, it does not optimize communications (see more details in Section 2.2.5, page 60).

Lemma 14 *After partitioning the shortest path matrix with the irregular partitioning technique and applying the snake-like mapping, the maximum number of blocks per processor is* $(3k+2)$. *k is the number of fragmentations performed.*

Proof. Depending on the parity of $f(p)$, the following two scenarios can happen:

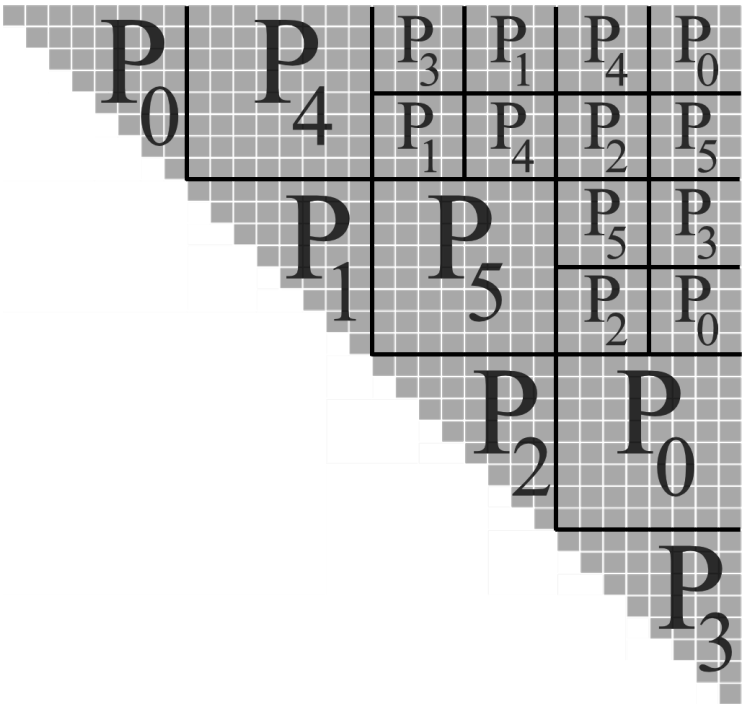


Figure 42 – *Snake-like mapping on six processors when* $k = 1$

1 - when $f(p)$ is odd, each processor has to evaluate at most one larger-size block, $3(k-1)$ blocks in the diagonals from the first to the $(k-1)$th fragmentation and 4 blocks after the kth fragmentation; thus $1+3(k-1)+4 = (3k+2)$ blocks in total;

2 - when $f(p)$ is even, each processor has to evaluate at most two larger-size blocks, $2(k-1)$ blocks in the diagonals from the first to the $(k-1)$th fragmentation and 3 blocks after the kth fragmentation; thus $2+2(k-1)+3 = (2k+3)$ blocks in total.

Since $3k+2 \geq 2k+3$ when $k \geq 1$, then each processor has to evaluate at most $(3k+2)$ blocks. ■

3.3.3 - CGM-based parallel algorithm for solving the MPP

To solve the MPP, our CGM-based parallel algorithm is a succession of $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ similar steps in which the blocks are evaluated in a progressive fashion as in (Kechid and Myoupo, 2009; Kengne and Myoupo, 2012). Thus, evaluating the shortest path cost of nodes of a block belonging to the diagonal d starts at the diagonal $\lceil d/2 \rceil$. The overall structure is given by Algorithm 21. After the computation of blocks on diagonal d (line 3 in Algorithm 21), each block is forwarded (line 4 in Algorithm 21) to processors that need these blocks for updating

Algorithm 21 Our CGM-based parallel algorithm based on the irregular partitioning technique to solve the MPP

1: $maxDiag \leftarrow f(p) + k \times (\lceil f(p)/2 \rceil + 1)$;
2: **for** $d = 1$ to $maxDiag$ **do**
3: **Finalization** of computations of the shortest path costs to nodes in blocks belonging to the diagonal d;
4: **Communication** of block $SM(i, j)$ of current diagonal to processors that hold upper and right blocks;
5: **Update** the shortest path costs to each block belonging to diagonals $(d+1, d+2, \ldots, \min\{2 \times (d-1), maxDiag\})$;

Algorithm 22 Finalization phase using in Algorithm 21 to evaluate blocks $SM(i, j)$ belonging to the lth level of fragmentation

1: **for** $d = (j - i - \theta(n, p, l))$ to $(j - i)$ **do**
2: **for each** node (a, b) of diagonal d belonging to $SM(i, j)$ **do**
3: $SP[a, b] \leftarrow \min\{SP[a, b]$, weight of paths whose final edge are jumps coming from block $SM(i, i + \theta(n, p, l))$, weights of paths whose final edges are internal jumps, weights of paths whose final edges are unit edges$\}$;

Algorithm 23 Updating phase using in Algorithm 21 to refresh the shortest path costs to nodes of blocks $SM(i, j)$ belonging to the lth level of fragmentation

1: **for** $d = (j - i - \theta(n, p, l))$ to $(j - i)$ **do**
2: $M_1 \leftarrow$ matrix-multiplication$(+, \min)$ $(SM(i, h + i - 1), SM(h + i - 1, j))$;
3: $M_2 \leftarrow$ matrix-multiplication$(+, \min)$ $(SM(i, j - h + 1), SM(j - h + 1, j))$;
4: $SP[i, j] \leftarrow \min\{SP[i, j], M_1, M_2\}$;

(line 5 in Algorithm 21) or for finalizing (line 3 in Algorithm 21) the computations of values in next steps. According to Theorem 3, it is deducible that the updates for a block $SM(i, j)$ belonging to the lth level of fragmentation are equivalent to a matrix multiplication $(+, \min)$ of matrices $SM(i, k)$ and $SM(k - \theta(n, p, l) + 2, j)$. Algorithms 22 and 23 show the pseudocodes of finalization and updating phases in Algorithm 21.

Theorem 13 *Our CGM-based parallel solution based on the irregular partitioning technique requires* $O\left(n^3/p\right)$ *execution time with* $\lceil \sqrt{2p} \rceil + k \times \left(\left\lceil \frac{\lceil \sqrt{2p} \rceil}{2} \right\rceil + 1 \right)$ *communication rounds in the worst case to solve the MPP.* k *is the number of fragmentations performed.*

Proof. Let $S = f(p) = \lceil 2p \rceil$. During the computation rounds, evaluating each block of a diagonal belonging to the lth level of fragmentation through the sequential algorithm of Godbole (1973) (used in Algorithm 22) and the sequential multiplication of two matrices (used in Algorithm 23) requires $O\left(\frac{n^3}{2^{3l} \times (2p)^{3/2}}\right) =$

Algorithm 24 Our CGM-based parallel algorithm based on the irregular partitioning technique to solve the OBST problem

1: **for** $d = 1$ to $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ **do**
2: **Computation** of blocks belonging to the round d using Algorithm 11;
3: **Communication** of entries (*Tree* and *Cut* tables) required for computing each block of rounds $\{d+1, d+2, \ldots, f(p) + k \times (\lceil f(p)/2 \rceil + 1)\}$;

$O\left(\frac{n^3}{8^l \times p\sqrt{p}}\right)$ local computation time. So, from the proof of Lemma 13, the evaluation of each diagonal of blocks required :

$$\begin{aligned} D &= O\left(\frac{n^3}{p\sqrt{p}}\right) \times \left\lfloor \frac{S}{2} \right\rfloor + O\left(\frac{n^3}{8 \times p\sqrt{p}}\right) \times \left(\left\lceil \frac{S}{2} \right\rceil + 1\right) + O\left(\frac{n^3}{8^2 \times p\sqrt{p}}\right) \times \\ &\left(\left\lceil \frac{S}{2} \right\rceil + 1\right) + \cdots + O\left(\frac{n^3}{8^k \times p\sqrt{p}}\right) \times \left(\left\lceil \frac{S}{2} \right\rceil + 1\right) + O\left(\frac{n^3}{8^k \times p\sqrt{p}}\right) \times \left\lceil \frac{S}{2} \right\rceil \\ &= O\left(\frac{n^3}{p}\right) + O\left(\frac{n^3}{p\sqrt{p}}\right) \times \left(\left\lceil \frac{S}{2} \right\rceil + 1\right) \left[\frac{1}{8} + \frac{1}{8^2} + \cdots + \frac{1}{8^k}\right] + O\left(\frac{n^3}{8^k \times p}\right) \\ &= O\left(\frac{n^3}{p}\right) \end{aligned}$$

Therefore, this algorithm requires $O\left(n^3/p\right)$ execution time. The number of communication rounds is derived from Lemma 13. ■

Remark 3 *When $k = 0$, our CGM-based parallel solution based on the irregular partitioning technique to solve the MPP is reduced to the one in (Kengne and Myoupo, 2012), running in $O\left(n^3/p\right)$ execution time with $\lceil \sqrt{2p} \rceil$ communication rounds.*

3.3.4 - CGM-based parallel algorithm for solving the OBST problem

To solve the OBST problem, our CGM-based parallel algorithm evaluates the values of shortest paths to each node of a block, starting from the first diagonal of blocks to the diagonal $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$, by using the sequential algorithm of Knuth (1971) in computation rounds; although it does not allow predicting before the beginning of the computation of the block $SM(i, j)$, which will be the values necessary for its evaluation. This is why the blocks are evaluated in a non-progressive fashion, that is, the evaluation of blocks of the diagonal d start after computing blocks of the diagonal $(d-1)$. Algorithm 24 draws a big picture.

Theorem 14 *Our CGM-based parallel solution based on the irregular partitioning technique requires* $O\left(n^2/\sqrt{p}\right)$ *execution time with* $\lceil\sqrt{2p}\rceil + k \times \left(\left\lceil\frac{\lceil\sqrt{2p}\rceil}{2}\right\rceil + 1\right)$ *communication rounds in the worst case to solve the OBST problem.* k *is the number of fragmentations performed.*

Proof. It is deducible from Theorem 13. ■

Remark 4 *When* $k = 0$*, our CGM-based parallel solution based on the irregular partitioning technique to solve the OBST problem is reduced to the one in (Myoupo and Kengne, 2014b), running in* $O\left(n^2/\sqrt{p}\right)$ *execution time with* $\lceil\sqrt{2p}\rceil$ *communication rounds.*

3.3.5 - Experimental results

Experimental setups

This section outlines experimental results of our CGM-based parallel solutions to solve the MPP and the OBST problem[1], and compares them with the best previous solutions (Kengne and Myoupo, 2012; Myoupo and Kengne, 2014b). These algorithms have been executed on the MatriCS platform (described in Section 2.4.7). The results are presented following the different values of (n, p, k), where :

- n is the data size, with values in the set $\langle$4096, 8192, 12288, 16384, 20480, 24576, 28672, 32768, 36864, 40960$\rangle$. For the MPP, we have used a random data set generated for the TCP problem in Section 2.4.7. For the OBST problem, we have generated the frequencies for each data size.
- p is the number of processors, with values in the set $\langle$1, 32, 64, 96, 128$\rangle$. When $p = 1$, the sequential algorithm of Godbole (1973) is carried out to solve the MPP and the sequential algorithm of Knuth (1971) to solve the OBST problem.
- k is the number of fragmentations performed, with values in the set $\langle$0, 1, 2, 3, 4, 5$\rangle$. When $k = 0$, our CGM-based parallel solutions solving the MPP and the OBST problem are similar to the one in (Kengne and Myoupo, 2012) and the one in (Myoupo and Kengne, 2014b), respectively (see Remarks 3 and 4).

Note 2 *These parameters will be used in the next CGM-based parallel solutions described in Sections 3.4 and 3.5.*

1. The source codes are available on these URLs : https://github.com/compiii/CGM-Sol-for-MPP and https://github.com/compiii/CGM-Sol-for-OBST. They implement all the solutions proposed in this chapter.

Tables 3 and 5 (respectively Tables 4 and 6) show the total execution time, the speedup, and the efficiency of our CGM-based parallel solutions to solve the MPP (respectively the OBST problem). Figures 43a, 43b, 43c, 43d, 44a, 44b, 45a, 45b, 46a, 46b, 46c, and 46d are drawn from the results obtained in Tables 3, 4, 5, and 6.

Evolution of the global communication time

Figures 43a, 43b, 43c, and 43d illustrate the global communication time, which is composed of the latency time of processors and the effective transfer time of data. In general, the global communication time is higher while solving the MPP compared to the one that is obtained while solving the OBST problem because evaluating blocks using the sequential algorithm of Knuth (1971) minimizes the latency time of processors, since a processor waits less time to start or continue evaluating a block. However, these figures show that the global communication time gradually decreases as the number of fragmentations increases. For example, on thirty-two processors when $n = 40960$, Figures 43a and 43c (respectively Figures 43b and 43d) reveal that the global communication time is reduced down to 34.98% (respectively 3.87%) when $k = 1$, and to 45.63% (respectively 40.49%) when $k = 2$ for the MPP (respectively the OBST problem). When a fragmentation is performed, the number of blocks increases by dividing the current size of blocks into four to form the smaller-size blocks. It allows minimizing the latency time of processors and the effective transfer time of data since the smaller-size blocks take less time to evaluate and communicate compared to the larger-size blocks.

Evolution of the load-balancing of processors

Figures 44a and 44b compare the load imbalance on thirty-two processors. The values are obtained by performing the difference between the average computation time of processors and the lowest (and highest) computation time among them. The processor with the lowest or highest computational load varies with the number of fragmentations for the MPP; for example, when $k = 0$ and $k = 2$, respectively, P_5 and P_0 have the lowest loads, and P_2 and P_{13} have the highest. However, for the OBST problem, the speedup of Knuth (1971) does not allow obtaining the same result as the MPP (yet it is the same dependency graph and the partitioning techniques are also the same). It is due to the fact that the blocks in the same diagonal have not the same load; for example, when $k = 0$ and $k = 2$, respectively, P_7 and P_{30} have the lowest loads, and P_3 and P_{22} have the highest.

These figures show that the irregular partitioning technique of the dynamic graph balances the load of processors better than the regular partitioning technique

Table 3 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 40960\}$, $p \in \{1, 32\}$, and $k \in \{0, 1, 2\}$ while solving the MPP with the irregular partitioning technique*

	Total execution time				Speedup			Efficiency		
n	$p = 1$	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$
4096	212.84	32.19	19.29	18.54	6.61	11.03	11.48	20.66	34.48	35.87
8192	2238.52	285.07	162.02	157.90	7.85	13.82	14.18	24.54	43.18	44.30
12288	8051.53	982.42	525.23	494.04	8.20	15.33	16.30	25.61	47.90	50.93
16384	19423.55	2728.25	1435.73	1400.95	7.12	13.53	13.86	22.25	42.28	43.33
20480	37932.72	6055.91	2668.47	2546.92	6.26	14.22	14.89	19.57	44.42	46.54
24576	65559.28	11607.89	5067.28	5036.99	5.65	12.94	13.02	17.65	40.43	40.67
28672	104688.35	19221.79	8609.44	8471.21	5.45	12.16	12.36	17.02	38.00	38.62
32768	172834.08	28806.13	14448.27	13071.16	6.00	11.96	13.22	18.75	37.38	41.32
36864	243774.87	41851.26	25484.45	17496.95	5.82	9.57	13.93	18.20	29.89	43.54
40960	383976.51	58074.15	34112.16	25867.56	6.61	11.26	14.84	20.66	35.18	46.39

Table 4 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 40960\}$, $p \in \{1, 32\}$, and $k \in \{0, 1, 2\}$ while solving the OBST problem with the irregular partitioning technique*

	Total execution time				Speedup			Efficiency		
n	$p = 1$	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$
4096	62.34	20.58	16.68	13.11	3.03	3.74	4.75	9.46	11.68	14.85
8192	514.32	180.82	136.43	100.02	2.84	3.77	5.14	8.89	11.78	16.07
12288	1784.19	621.52	476.46	343.27	2.87	3.74	5.20	8.97	11.70	16.24
16384	4315.43	1582.57	1142.21	818.11	2.73	3.78	5.27	8.52	11.81	16.48
20480	8546.86	2919.91	2308.19	1600.75	2.93	3.70	5.34	9.15	11.57	16.69
24576	14899.60	5970.20	3920.49	2785.83	2.50	3.80	5.35	7.80	11.88	16.71
28672	23802.78	8527.82	6078.21	4428.91	2.79	3.92	5.37	8.72	12.24	16.80
32768	33263.71	12533.26	10130.78	6607.36	2.65	3.28	5.03	8.29	10.26	15.73
36864	48356.46	17658.48	13514.64	9451.97	2.74	3.58	5.12	8.56	11.18	15.99
40960	69722.12	24910.11	20933.65	12986.77	2.80	3.33	5.37	8.75	10.41	16.78

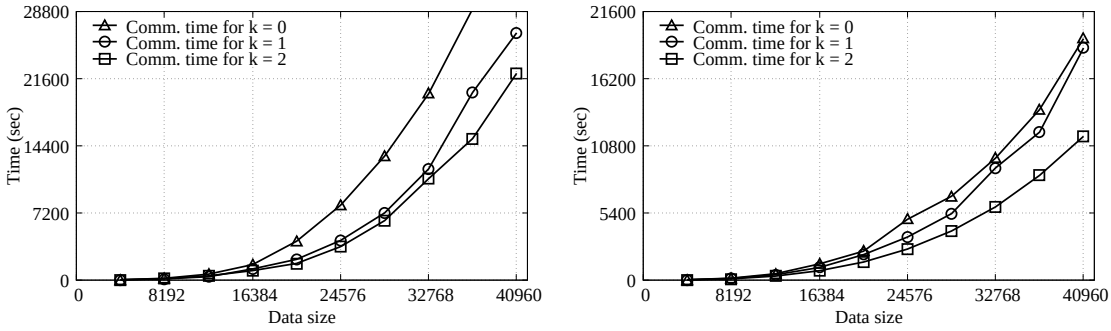
Table 5 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{0, 1, 2\}$ while solving the MPP with the irregular partitioning technique*

	Total execution time			Speedup			Efficiency		
p	$k=0$	$k=1$	$k=2$	$k=0$	$k=1$	$k=2$	$k=0$	$k=1$	$k=2$
$n = 36864$									
64	19656.70	9584.39	8411.20	12.40	25.43	28.98	19.38	39.74	45.28
96	13729.69	7204.86	5995.95	17.76	33.83	40.66	18.50	35.24	42.35
128	10133.01	5393.59	4565.67	24.06	45.20	53.39	18.79	35.31	41.71
$n = 40960$									
64	28392.16	13798.22	11932.76	13.52	27.83	32.18	21.13	43.48	50.28
96	20024.65	10160.46	8904.22	19.18	37.79	43.12	19.97	39.37	44.92
128	14961.59	7917.73	6785.92	25.66	48.50	56.58	20.05	37.89	44.21

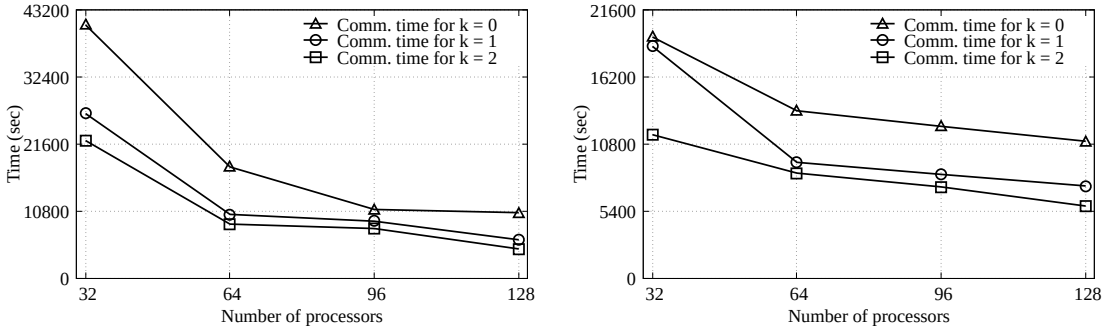
Table 6 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{0, 1, 2\}$ while solving the OBST problem with the irregular partitioning technique*

	Total execution time			Speedup			Efficiency		
p	$k=0$	$k=1$	$k=2$	$k=0$	$k=1$	$k=2$	$k=0$	$k=1$	$k=2$
$n = 36864$									
64	11671.40	7610.84	6586.99	4.14	6.35	7.34	6.47	9.93	11.47
96	10128.02	6573.25	5552.72	4.77	7.36	8.71	4.97	7.66	9.07
128	8951.40	5788.29	5126.66	5.40	8.35	9.43	4.22	6.53	7.37
$n = 40960$									
64	16045.47	10456.06	9057.29	4.35	6.67	7.70	6.79	10.42	12.03
96	13928.60	9031.15	7813.92	5.01	7.72	8.92	5.21	8.04	9.29
128	12305.95	7949.95	6707.94	5.67	8.77	10.39	4.43	6.85	8.12

of this graph due to the gradual reduction of the block sizes that allows processors to remain active as long as possible. Indeed, the fragmentation is performed on the blocks with the largest loads. So, some small-size blocks (which are in the upper diagonals) have higher loads than the large ones (which are in the lower diagonals). Thanks to the snake-like mapping, which allows distributing blocks onto processors in an equitable way, a processor can have in the worst case one more block than another. This greatly contributes to balancing the loads of processors. When $n = 40960$ in Figure 44a, the lowest load and the highest load respectively decrease on average by 25.95% and 43.72% when $k = 1$, and on average by 51.14% and 62.29% when $k = 2$. Similar observations can be made in Figure 44b, except when $n = 40960$ and $k = 1$ where the lowest load narrows down to 27.52% and the highest load inceases up to 11.89% because of the speedup of Knuth (1971). However when $k = 2$, the lowest load and the highest load reduce down to 56.63% and 42.98%.

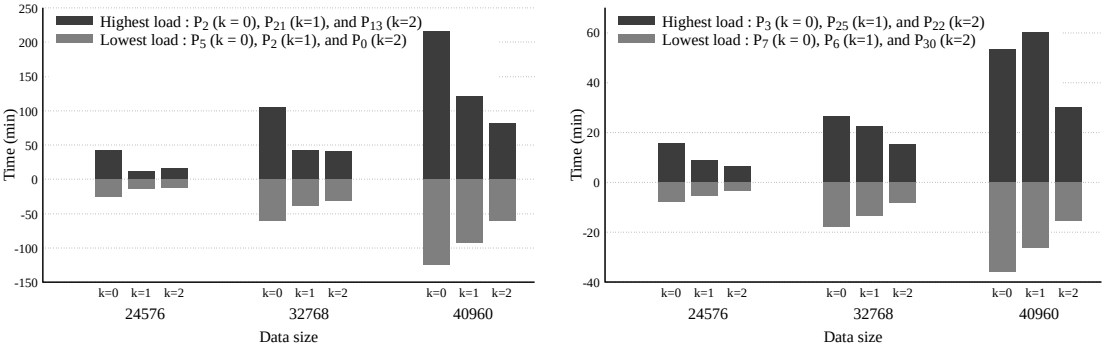


(a) *MPP for* $p = 32$ **(b)** *OBST problem for* $p = 32$



(c) *MPP for* $n = 40960$ **(d)** *OBST problem for* $n = 40960$

Figure 43 – *Global communication time for* $n \in \{4096, \ldots, 40960\}$, $p \in \{32, \ldots, 128\}$, *and* $k \in \{0,1,2\}$ *while solving the MPP and the OBST problem with the irregular partitioning technique*

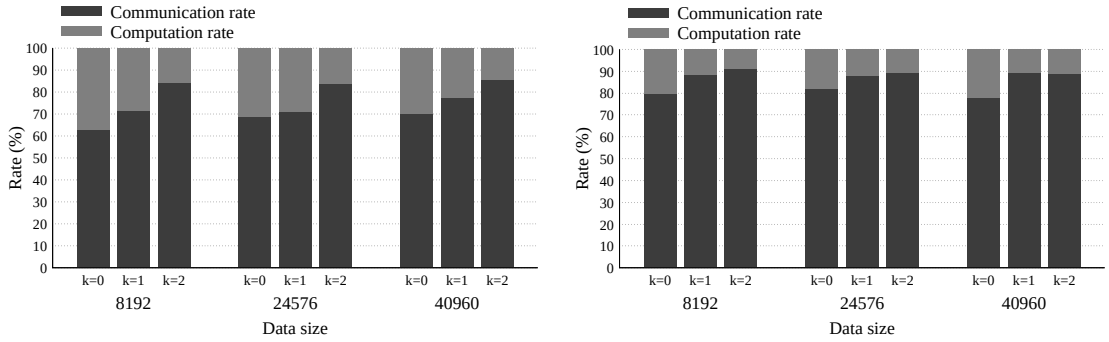


(a) *MPP* **(b)** *OBST problem*

Figure 44 – *Load imbalance of processors for* $n \in \{24576, 32768, 40960\}$, $p = 32$, *and* $k \in \{0,1,2\}$ *while solving the MPP and the OBST problem with the irregular partitioning technique*

Comparison of communication rate and computation rate

Figures 45a and 45b show that the communication rate is higher than the computation rate, which means that the global communication time is a significant part of the total execution time compared to the overall computation time whatever the data size and the number of fragmentations on thirty-two processors.



(a) *MPP* **(b)** *OBST problem*

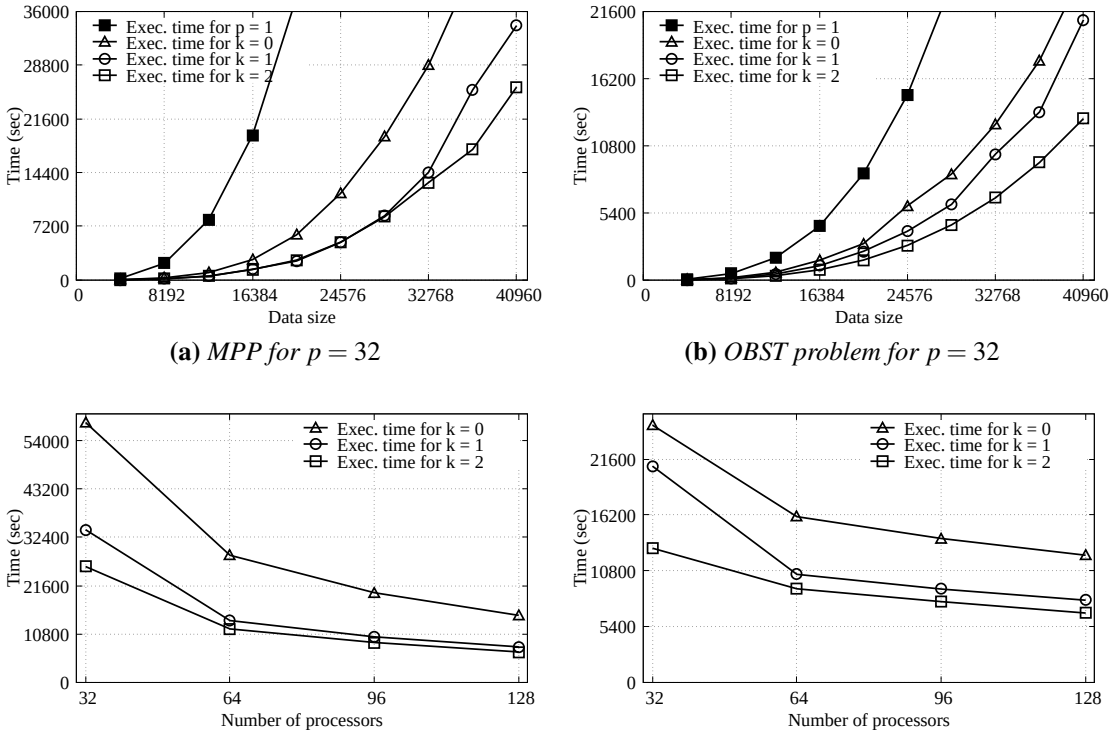
Figure 45 – *Computation rate versus communication rate for* $n \in \{8192, 24576, 40960\}$, $p = 32$, *and* $k \in \{1, 2\}$ *while solving the MPP and the OBST problem with the irregular partitioning technique*

When $n = 40960$, the computation rate and the communication rate respectively are 29.78% and 70.22% when $k = 0$, 22.28% and 77.72% when $k = 1$, and 14.30% and 85.70% when $k = 2$ for the MPP; and respectively are 22.05% and 77.95% when $k = 0$, 10.83% and 89.17% when $k = 1$, and 11.01% and 88.99% when $k = 2$ for the OBST problem. It is therefore deducible from this observation that the irregular partitioning technique reduces the overall computation time because the computation rate is lower and lower when the number of fragmentations increases, since the computational load of processors is more and more balanced.

Evolution of the total execution time

Figures 46a, 46b, 46c, and 46d show that the minimization of the overall communication and computation time lead up to a reduction of the total execution time as the number of fragmentations increases. For example, on thirty-two processors when $n = 40960$ in Figures 46a and 46c, the total execution time decreases on average by 41.26% when $k = 1$, and on average by 55.46% when $k = 2$ for the MPP; in Figures 46b and 46d, the total execution time decreases on average by 15.96% when $k = 1$, and on average by 47.87% when $k = 2$ for the OBST problem. Similar observations can be made on one hundred and twenty-eight processors. The total execution time decreases on average by 47.08% when $k = 1$, and on average by 54.64% when $k = 2$ for the MPP; and it decreases on average by 35.40% when $k = 1$, and on average by 45.50% when $k = 2$ for the OBST problem.

Tables 3, 4, 5, and 6 show that since the total execution time decreases when the number of fragmentations increases, the speedup and the efficiency of our CGM-based parallel solutions increase. On thirty-two processors when $n = 40960$ in Table 3 (respectively Table 4), the speedup and the efficiency are equal to 6.61 and



(a) *MPP for* $p = 32$ **(b)** *OBST problem for* $p = 32$

(c) *MPP for* $n = 40960$ **(d)** *OBST problem for* $n = 40960$

Figure 46 – *Total execution time for* $n \in \{4096, \ldots, 40960\}$*,* $p \in \{32, \ldots, 128\}$*, and* $k \in \{0, 1, 2\}$ *while solving the MPP and the OBST problem with the irregular partitioning technique*

20.66% (respectively 2.80 and 8.75%) when $k = 0$, and increase up to 11.26 and 35.18% (respectively 3.33 and 10.41%) when $k = 1$, and up to 14.84 and 46.39% (respectively 5.37 and 16.78%) when $k = 2$ for the MPP (respectively the OBST problem). From all this, we can deduce that our CGM-based parallel solutions based on the irregular partitioning technique are scalable as the data size, the number of processors, and the number of fragmentations rise.

3.3.6 - Drawback of the irregular partitioning technique

Although the irregular partitioning technique has good performance, it suffers from an important shortcoming. Indeed, our previous CGM-based parallel solutions do not allow processors to start evaluating small-size blocks as soon as the data they need are available. Yet, these data are usually available before the end of the evaluation of large-size blocks. Figure 47 briefly illustrates this drawback. The processors colored in yellow (P_3 and P_7) evaluate the blocks that have been assigned to them. These blocks need the data (more precisely the most shaded blocks and subblocks) from the processors colored in green (P_0, P_1, P_2, P_4, P_5, and P_6) to start or continue their evaluations. P_3 (respectively P_7) requires the data from P_0, P_1,

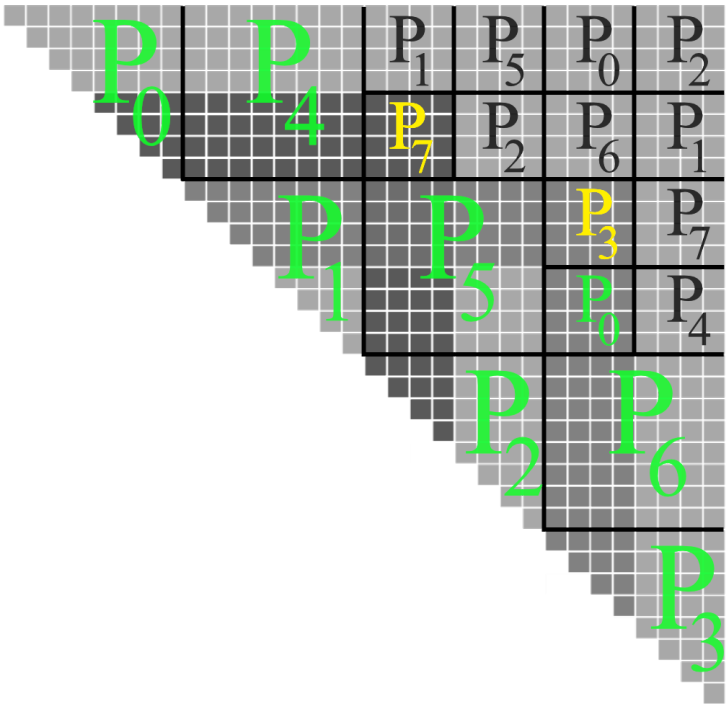


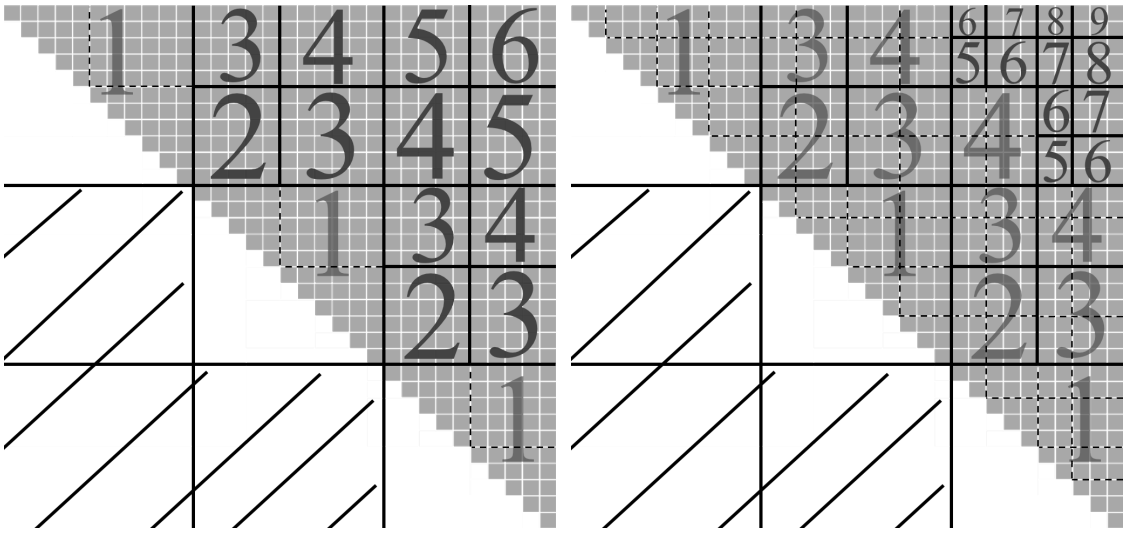
Figure 47 – *Drawback of the irregular partitioning technique*

P_5, and P_6 (respectively P_0, P_2, P_4, and P_5). However, these processors holding blocks on which P_3 and P_7 depend must completely finish their evaluations before communicating them entirely or partially (see Remark 2, page 109) to P_3 and P_7. As a result, this shortcoming induces an important latency time of processors, which is the largest part of the global communication time.

3.4 - Second dynamic graph partitioning : k-block splitting technique

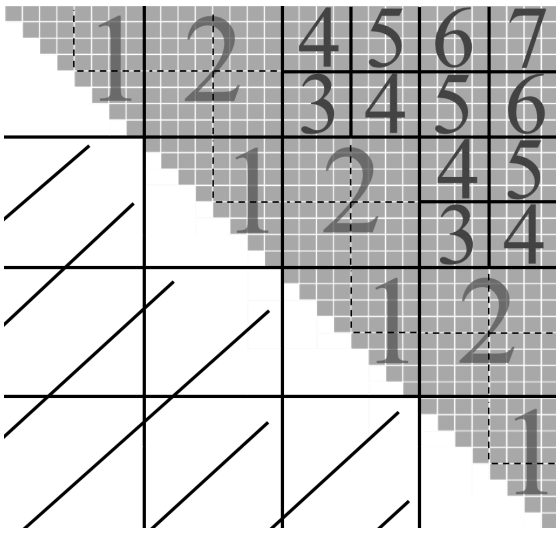
The k-block splitting technique aims to reduce the latency time of processors by allowing them to start the evaluation of blocks as soon as possible. The goal is to give the possibility to processors which depend on them not to wait too much. This strategy consists in splitting the large-size blocks into a set of smaller-size blocks called *k-blocks* after performing k fragmentations. Thus, evaluating a block by a single processor will consist of computing and communicating each k-block contained in this block. It will allow processors to start the evaluation of k-blocks as soon as the data they need will available.

Let $f(p) = \lceil \sqrt{2p} \rceil$, $\theta(n,p) = \left\lceil \frac{n}{f(p)} \right\rceil$, and $\theta(n,p,l) = \left\lceil \frac{\theta(n,p)}{2^l} \right\rceil$. Formally, we partition the shortest path matrix SP into blocks (denoted by $SM(i,j)$), and split

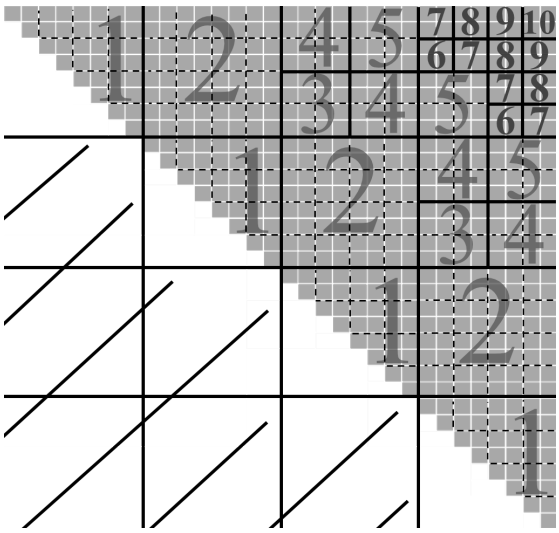


(a) $p \in \{3, 4\}$ *and* $k = 1$

(b) $p \in \{3, 4\}$ *and* $k = 2$



(c) $p \in \{5, 6, 7, 8\}$ *and* $k = 1$



(d) $p \in \{5, 6, 7, 8\}$ *and* $k = 2$

Figure 48 – *k-block splitting technique of the shortest path matrix for* $n = 32$, $k \in \{1,2\}$, *and* $p \in \{3, 4, 5, 6, 7, 8\}$. *For* $p \in \{3, 4\}$, *SP is partitioned into fifteen blocks and twenty-one k-blocks when* $k = 1$, *and into twenty-four blocks and seventy-eight k-blocks when* $k = 2$. *For* $p \in \{5, 6, 7, 8\}$, *SP is partitioned into nineteen blocks and thirty-six k-blocks when* $k = 1$, *and into twenty-eight blocks and one hundred and thirty-six when* $k = 2$

the large-size blocks into a set of k-blocks. $SM(i, j)$ is thus a $\theta(n, p, l) \times \theta(n, p, l)$ matrix belonging to the lth level of fragmentation and is subdivided into 4^{k-l} k-blocks of size $\theta(n, p, k) \times \theta(n, p, k)$. Figures 48a, 48b, 48c, and 48d depict four scenarios of this partitioning for $n = 32$, $k \in \{1,2\}$, and $p \in \{3, 4, 5, 6, 7, 8\}$.

Remark 5 *About this partitioning, some points are similar to Remark 1. We enumerate the most relevant ones :*

1 - the blocks of the first diagonal are $\theta(n,p) \times \theta(n,p)$ *upper triangular matrices splitting into* $2^{k-1}(2^k+1)$ *k-blocks; this is illustrated for example in Figures 48b and 48d where, after performing two fragmentations (i.e.* $k=2$*), the upper triangular matrices are subdivided into ten k-blocks;*

2 - a block belonging to the lth level of fragmentation is full if it is a $\theta(n,p,l) \times \theta(n,p,l)$ *non-triangular matrix;*

3 - one fragmentation increases up to $(\lceil f(p)/2 \rceil + 1)$ *the number of diagonal of blocks (see proof in Lemma 13);*

4 - there are $2^k \times f(p)$ *diagonals of k-blocks after performing k fragmentations. This is illustrated for example in Figures 48c and 48d where there respectively are eight diagonals of k-blocks when* $k=1$ *and sixteen diagonals of k-blocks when* $k=2$*.*

The purpose of splitting the blocks into a set of k-blocks is to progressively evaluate and communicate them during the evaluation of blocks. The k-block splitting technique is essentially based on the progressive evaluation of the DAG nodes (see Theorem 3, page 57). It is therefore adequate to solve the MPP because the sequential algorithm of Godbole (1973) gives the possibility to evaluate the nodes in this way. In contrast, the sequential algorithm of Knuth (1971) does not allow performing this kind of evaluation to solve the OBST problem; this is because the speedup of Knuth (1971) does not allow knowing when to start or continue the evaluation of a node. Thus, it would not be meaningful and practical to apply the k-block splitting technique to solve the OBST problem because a lot of unnecessary computations could be performed and lead to poor performance.

In summary, the k-block splitting technique is used in this section to solve the MPP, which is the most classical and generic problem. Blocks' dependency analysis, CGM-based parallel algorithms derived from this technique, and experimental results are presented in Sections 3.4.1, 3.4.2, and 3.4.3 respectively.

3.4.1 - Blocks' dependency analysis of the MPP

From the blocks' dependency seen in Section 3.3.1 for the irregular partitioning technique, it is easy to analyze the k-blocks dependency. Indeed, if the blocks on the same diagonal are independent, then the parts of these blocks, i.e. the k-blocks, will also be independent. Figure 49 depicts an example of dependencies of two k-blocks $SM(i,j)$ and $SM(h,l)$ after applying the k-block splitting technique. They require the already computed values of the most shaded k-blocks, which are part

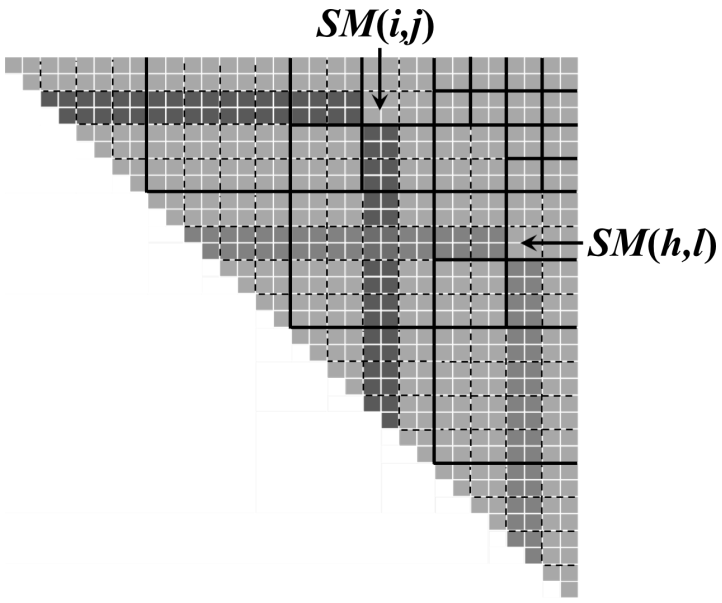


Figure 49 – *Dependencies of two k-blocks $SM(i,j)$ and $SM(h,l)$ after applying the k-block splitting technique*

or not of a same block. Evaluating the shortest paths of nodes of k-blocks can be then carried out in parallel.

3.4.2 - CGM-based parallel algorithms to solve the MPP

Depending on how processors evaluate the k-blocks contained in a block, two approaches can be derived : the diagonal by diagonal evaluation approach and the k-block by k-block evaluation approach. The overall structure of our CGM-based parallel algorithms is similar to Algorithm 2, which is based on the irregular partitioning technique, since $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ diagonal of blocks must be evaluated in a progressive fashion whatever the chosen approach.

Diagonal by diagonal evaluation approach

The diagonal by diagonal evaluation approach consists in evaluating a block by computing a set of k-blocks belonging to the same diagonal of k-blocks before communicating them. This approach will minimize the number of communication rounds. However, it will not allow processors to evaluate other k-blocks as soon as the data they need are available. In fact, after performing a k-block, a processor will compute other k-blocks of the same diagonal before communicating them.

This approach is given by Algorithm 25. Evaluating the shortest path to a node of a k-block of diagonal d starts at diagonals $\lceil d/2 \rceil$. The number of diagonal of k-blocks goes from left to right and ranges from 1 to $2^k \times f(p)$. At the end of the computation of k-blocks on diagonal d (line 4 in Algorithm 25), each k-block

Algorithm 25 Our CGM-based parallel algorithm based on the k-block splitting technique to solve the MPP using the diagonal by diagonal evaluation approach

1: $maxDiag \leftarrow f(p) + k \times (\lceil f(p)/2 \rceil + 1)$;
2: **for** $u = 1$ to $maxDiag$ **do**
3: **for** $d = \lceil (j-i)/\theta(n,p,k) \rceil - 2^{k-l} - 2$ to $\lceil (j-i)/\theta(n,p,k) \rceil$ **do**
4: **Finalization** of the evaluation of the block $SM(i,j)$ of diagonal u : compute the shortest path costs to nodes of each k-block of the same diagonal d;
5: **Communication** of each k-blocks belonging to the same diagonal d to processors that detain upper blocks and right blocks;
6: **for** $d = \lceil (j'-i')/\theta(n,p,k) \rceil - 2^{k-l'} - 2$ to $\lceil (j'-i')/\theta(n,p,k) \rceil$ **do**
7: **Update** the shortest path costs to nodes of each k-blocks contained in the block $SM(i',j')$ of diagonals $(u+1, u+2, \ldots, \min\{2 \times (u-1), maxDiag\})$;

is forwarded (line 5 in Algorithm 25) to processors that need these k-blocks for updating (line 7 in Algorithm 25) or for finalizing (line 4 in Algorithm 25) the computations of values in next steps.

Theorem 15 *To solve the MPP, our CGM-based parallel solution based on the k-block splitting technique using the diagonal by diagonal evaluation approach runs in* $O\left(n^3/4^k p\right)$ *execution time with* $O\left(2^k\sqrt{p}\right)$ *communication rounds in the worst case.*

Proof. Let $S = f(p) = \lceil \sqrt{2p} \rceil$ and $\beta = (S \bmod 2)$. Algorithm 25 evaluates $S + k \times (\lceil S/2 \rceil + 1)$ diagonals of blocks. A single processor computes and communicates:

- 2^k diagonals of k-blocks in the first diagonal of blocks;
- $(2^{k+1} - 1)$ diagonals of k-blocks from the diagonal of blocks 2 to $(\lfloor S/2 \rfloor - 1)$;
- $(2^{l+1} - 1)$ diagonals of k-blocks for each $(\lceil S/2 \rceil + 1)$ diagonals of blocks belonging to the lth level of fragmentation such that $1 \leq l < k$;
- one diagonal of k-blocks for each $(S + \beta)$ diagonals of blocks belonging to the kth level of fragmentation.

Thus, the number of communication rounds is equal to :

$$\begin{aligned} D &= 2^k + \left(2^{k+1} - 1\right)\left(\left\lfloor \frac{S}{2} \right\rfloor - 1\right) + \left(\left\lceil \frac{S}{2} \right\rceil + 1\right)\sum_{l=1}^{k-1}\left(2^{l+1} - 1\right) + S + \beta \\ &= \left\lfloor \frac{S}{2} \right\rfloor \left(2^{k+1} - 1\right) + \left\lceil \frac{S}{2} \right\rceil \left(2^{k+1} - k - 3\right) + 2^k - k - 2 + S + \beta \\ &= O\left(2^k\sqrt{p}\right) \end{aligned}$$

Algorithm 26 Our CGM-based parallel algorithm based on the k-block splitting technique to solve the MPP using the k-block by k-block evaluation approach

1: $maxDiag \leftarrow f(p) + k \times (\lceil f(p)/2 \rceil + 1)$;
2: **for** $u = 1$ to $maxDiag$ **do**
3: **for each** k-block ρ belonging in the block $SM(i, j)$ **do**
4: **Finalization** of the evaluation of the block $SM(i, j)$ of diagonal u : compute the shortest path costs to nodes of the k-block ρ;
5: **Communication** of the k-block ρ to the processors that detain upper blocks and right blocks;
6: **for each** $\rho' \in SM(i', j')$ of diagonals $(u+1, \ldots, \min\{2 \times (u-1), maxDiag\})$ **do**
7: **Update** the shortest path costs to nodes of the k-block ρ';

Therefore, this algorithm runs in $O\left(n^3/4^k p\right)$ execution time since evaluating a k-block requires $O\left(n^3/8^k p\sqrt{p}\right)$ local computation time. ■

k-block by k-block evaluation approach

The k-block by k-block evaluation approach consists in evaluating a block by computing and communicating each k-block contained in this block. It will allow the processors to evaluate other k-blocks as soon as the data they need are available since a k-block is communicated as soon as a processor has finished computing it. However, compared to the first, this approach will involve a lot of communication between them. This approach is not different to the first one, except that we communicate a k-block immediately after computing it. Algorithm 26 gives an overview.

Theorem 16 *In the worst case, our CGM-based parallel solution based on the k-block splitting technique to solve the MPP using the k-block by k-block evaluation approach runs in $O\left(n^3/2^k p\right)$ execution time with $O\left(4^k\sqrt{p}\right)$ communication rounds.*

Proof. A single processor computes and communicates:

- $2^{k-1}(2^k - 1)$ k-blocks at the first diagonal of blocks;
- 4^k k-blocks from the diagonal of blocks 2 to $(\lfloor S/2 \rfloor - 1)$;
- 4^{k-l} k-blocks for each $(\lceil S/2 \rceil + 1)$ diagonals of blocks belonging to the lth level of fragmentation such that $1 \leq l < k$;
- one k-block for each $(S + \beta)$ diagonals of blocks belonging to the kth level of fragmentation.

Thus, the number of communication rounds is equal to :

$$D = 2^{k-1}(2^k - 1) + 4^k \left(\left\lfloor \frac{S}{2} \right\rfloor - 1 \right) + \left(\left\lceil \frac{S}{2} \right\rceil + 1 \right) \sum_{l=1}^{k-1} 4^{k-l} + S + \beta$$
$$= O\left(4^k \sqrt{p}\right)$$

Therefore, this algorithm runs in $O\left(n^3/2^k p\right)$ execution time. ■

3.4.3 - Experimental results

This section presents experimental results of our CGM-based parallel solutions based on the k-block splitting technique to solve the MPP, and compares them with the best previous solutions. Tables 7 and 8 show the total execution time, the speedup, and the efficiency of our CGM-based parallel solutions. In these tables, the solution using the diagonal by diagonal evaluation strategy is referred to *dbyd*, and the solution using the k-block by k-block evaluation strategy is referred to *kbyk*. Our previous CGM-based parallel solution based on the irregular partitioning technique (described in Sectoin 3.3) is referred to *frag*. Figures 50a, 50b, 51a, 51b, 52a, 52b, 53a, 53b, 53c, 53d, 54a, and 54b are drawn from the results obtained in Tables 3, 5, 7, 8, and 9.

Evolution of the global communication time

Figures 50a and 50b show that the global communication time decreases when the number of fragmentations increases. To minimize the latency time of processors, *frag* reduces the computation time of blocks by reducing their sizes. For example, on thirty-two processors when $n = 40960$, the communication time decreases on average by 34.98% when $k = 1$ and on average by 45.63% when $k = 2$. By allowing processors to receive the data they need as soon as they are available to start or continue computation, *dbyd* and *kbyk* minimize the global communication time better than *frag*. For example, on thirty-two processors when $n = 40960$, *dbyd* (respectively *kbyk*) reduces the global communication time on average by 47.67% (respectively 48.17%) when $k = 1$ and on average by 51.64% (respectively 56.97%) when $k = 2$. *kbyk* is better than *dbyd* because *kbyk* communicates a k-block as soon as a processor has finished computing it.

Evolution of the load-balancing of processors

Figures 51a and 51b show that on thirty-two processors, the load-balancing of processors for *frag*, *dbyd*, and *kbyk* are nearly the same. When $n = 40960$ and $k = 1$ (respectively $k = 2$), the lowest load and the highest load decrease on average by

Table 7 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 40960\}$, $p = 32$, and $k \in \{1,2\}$ while solving the MPP with the k-block splitting technique*

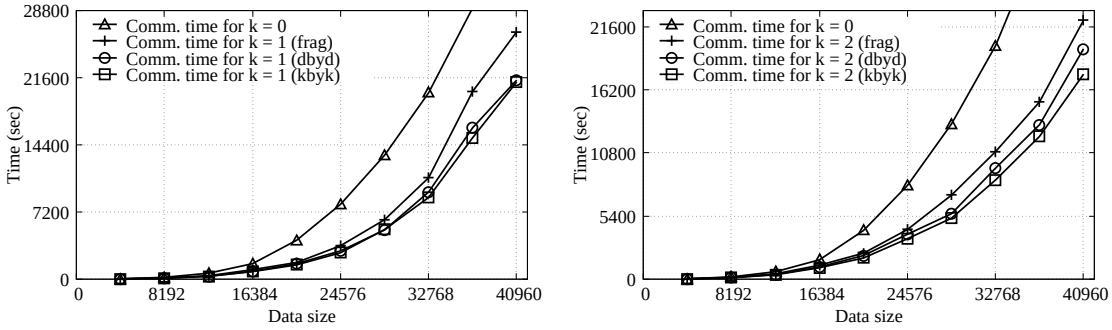
	Total execution time				Speedup				Efficiency			
n	$k=1$		$k=2$		$k=1$		$k=2$		$k=1$		$k=2$	
	dbyd	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*
4096	17.40	16.14	16.15	15.43	12.23	13.19	13.18	13.79	38.21	41.21	41.19	43.09
8192	146.61	135.66	138.70	131.94	15.27	16.50	16.14	16.97	47.71	51.56	50.44	53.02
12288	470.27	443.62	444.85	436.62	17.12	18.15	18.10	18.44	53.50	56.72	56.56	57.63
16384	1260.72	1220.34	1245.55	1177.73	15.41	15.92	15.59	16.49	48.15	49.74	48.73	51.54
20480	2428.76	2281.30	2407.38	2193.53	15.62	16.63	15.76	17.29	48.81	51.96	49.24	54.04
24576	4699.22	4292.97	4470.52	4205.99	13.95	15.27	14.66	15.59	43.60	47.72	45.83	48.71
28672	7502.89	7470.79	6692.89	6308.90	13.95	14.01	15.64	16.59	43.60	43.79	48.88	51.86
32768	12930.32	12343.33	11756.47	10725.43	13.37	14.00	14.70	16.11	41.77	43.76	45.94	50.36
36864	21796.13	20465.65	15565.01	14608.09	11.18	11.91	15.66	16.69	34.95	37.22	48.94	52.15
40960	29693.54	28875.73	23348.62	21175.60	12.93	13.30	16.45	18.13	40.41	41.55	51.39	56.67

Table 8 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{1,2\}$ while solving the MPP with the k-block splitting technique*

	Total execution time				Speedup				Efficiency			
p	$k=1$		$k=2$		$k=1$		$k=2$		$k=1$		$k=2$	
	dbyd	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*	*dbyd*	*kbyk*
$n = 36864$												
64	8399.46	7753.09	7279.01	6784.47	29.02	31.44	33.49	35.93	45.35	49.13	52.33	56.14
96	6289.94	5995.56	5053.26	4663.64	38.76	40.66	48.24	52.27	40.37	42.35	50.25	54.45
128	4782.50	4502.34	3831.61	3507.80	50.97	54.14	63.62	69.50	39.82	42.30	49.70	54.29
$n = 40960$												
64	12321.05	11724.24	10349.04	9485.96	31.16	32.75	37.10	40.48	48.69	51.17	57.97	63.25
96	8924.64	8569.33	7592.07	6931.47	43.02	44.81	50.58	55.40	44.82	46.68	52.68	57.70
128	6671.59	6317.01	5713.98	5163.69	57.55	60.78	67.20	74.36	44.96	47.49	52.50	58.09

Table 9 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n = 40960$, $p \in \{32, \ldots, 128\}$, and $k \in \{3,4\}$ while solving the MPP with the k-block splitting technique using the k-block by k-block evaluation strategy*

	Total execution time		Speedup		Efficiency	
p	$k=3$	$k=4$	$k=3$	$k=4$	$k=3$	$k=4$
32	29041.75	53490.34	13.22	7.18	41.32	22.43
64	11301.82	18374.91	33.97	20.90	53.09	32.65
96	7908.40	12164.26	48.55	31.57	50.58	32.88
128	5765.98	8729.30	66.59	43.99	52.03	34.36



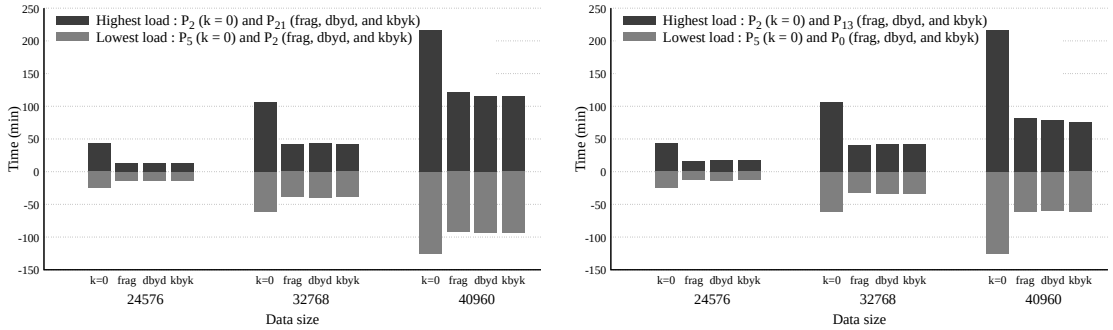
(a) $k = 1$ **(b)** $k = 2$

Figure 50 – *Global communication time for $n \in \{4096, \ldots, 40960\}$, $p = 32$, and $k \in \{0,1,2\}$ while solving the MPP with the k-block splitting technique*

25.95% and 43.72% (respectively 51.14% and 62.29%) for *frag*, on average by 25.51% and 46.92% (respectively 52.51% and 63.81%) for *dbyd*, and on average by 25.16% and 46.43% (respectively 50.59% and 64.90%) for *kbyk*. These results were predictable because *frag*, *dbyd*, and *kbyk* are essentially based on the irregular partitioning technique and snake-like mapping. Recall that the irregular partitioning technique balances the load of processors better than the regular partitioning technique because of the gradual reduction of the block sizes that allows processors to remain active as long as possible. From this, we can conclude that our CGM-based parallel solutions based on the *k*-block splitting technique (*dbyd* and *kbyk*) kept the good performance of our CGM-based parallel solution based on the irregular partitioning technique (*frag*) with respect to the load-balancing of processors. They promote the load balancing when the number of fragmentations rises.

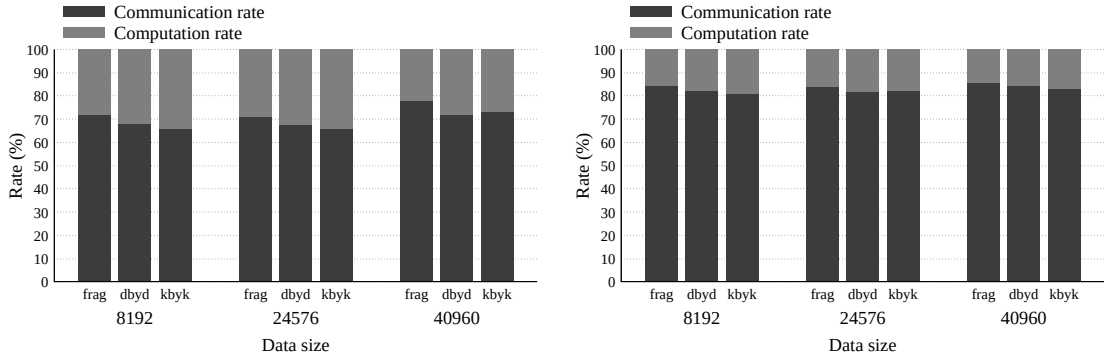
Comparison of communication rate and computation rate

Figures 52a and 52b show that the computation rate is lower than the communication rate on thirty-two processors. It was expected because the evaluation of a block belonging to the diagonal d starts before the step $(d-1)$ to minimize its computation time. The computation rate is lower and lower because when the number of



(a) $k = 1$ **(b)** $k = 2$

Figure 51 – *Load imbalance of processors for* $n \in \{24576, 32768, 40960\}$*,* $p = 32$*, and* $k \in \{0,1,2\}$ *while solving the MPP with the k-block splitting technique*



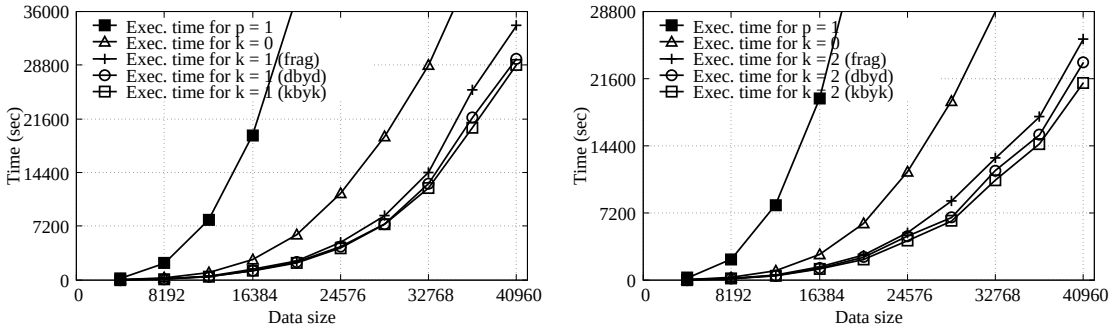
(a) $k = 1$ **(b)** $k = 2$

Figure 52 – *Computation rate versus communication rate for* $n \in \{8192, 24576, 40960\}$*,* $p = 32$*, and* $k \in \{1,2\}$ *while solving the MPP with the k-block splitting technique*

fragmentations increases, the computational load between processors is more and more balanced. When $n = 40960$ and $k = 1$ (respectively $k = 2$), it is 22.28% (respectively 14.30%) for *frag*, 28.14% (respectively 15.55%) for *dbyd*, and 26.82% (respectively 17.15%) for *kbyk*. Therefore, as for the load-balancing of processors, *dbyd* and *kbyk* kept the good performance of *frag*.

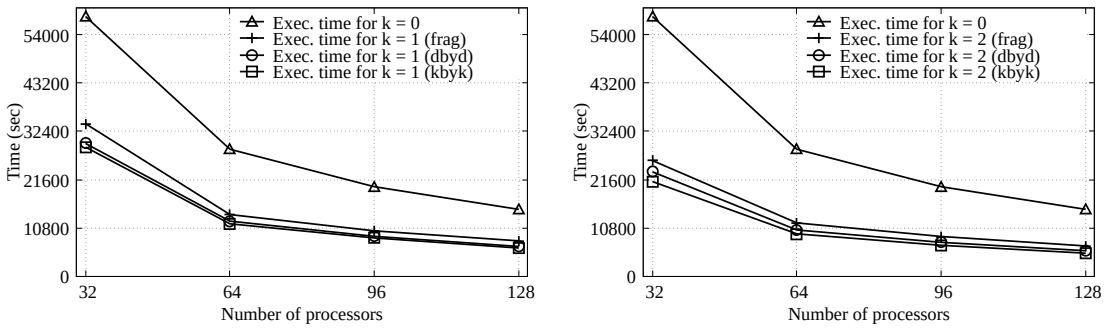
Evolution of the total execution time

Figures 53a, 53b, 53c, and 53d show that the minimization of the global communication time (due to the minimization of the latency time of processors) lead up to a reduction of the total execution time as the number of fragmentations rises. For example, on thirty-two processors when $n = 40960$ and $k = 1$ (respectively $k = 2$) in Figures 53a and 53b, the total execution time decreases on average by 41.26% (respectively 55.46%) for *frag*, on average by 48.87% (respectively 59.80%) for *dbyd*, and on average by 50.28% (respectively 63.54%) for *kbyk*. Similar observations can be made on one hundred and twenty-eight processors in Figures 53c



(a) $p = 32$ *and* $k = 1$

(b) $p = 32$ *and* $k = 2$



(c) $n = 40960$ *and* $k = 1$

(d) $n = 40960$ *and* $k = 2$

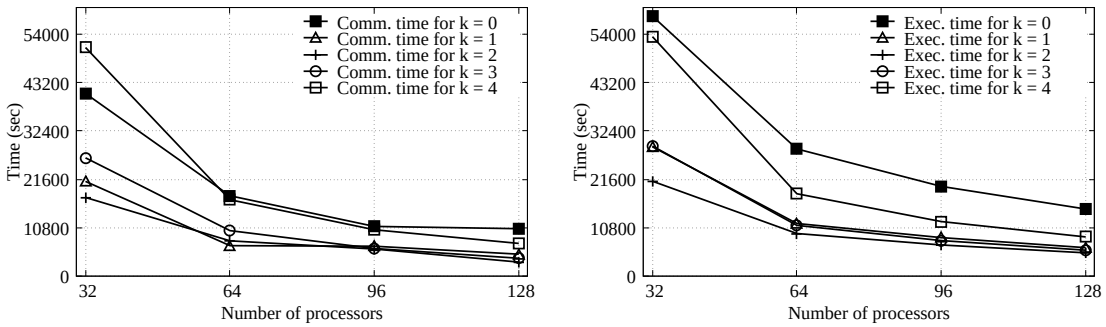
Figure 53 – *Total execution time for* $n \in \{4096, \ldots, 40960\}$*,* $p \in \{32, \ldots, 128\}$*, and* $k \in \{0, 1, 2\}$ *while solving the MPP with the k-block splitting technique*

and 53d; where when $n = 40960$ and $k = 1$ (respectively $k = 2$), the total execution time decreases on average by 47.08% (respectively 54.64%) for *frag*, on average by 55.41% (respectively 61.81%) for *dbyd*, and on average by 57.78% (respectively 65.49%) for *kbyk*.

Tables 7 and 8 show that the speedup and the efficiency of our CGM-based parallel solutions rise since the total execution time decreases as the number of fragmentations increases. On thirty-two processors when $n = 40960$ and $k = 1$ (respectively $k = 2$) in Table 7, the speedup and the efficiency are equal to 12.93 and 40.41% (respectively 16.45 and 51.39%) for *dbyd*, and increases up to 13.30 and 41.55% (respectively 18.13 and 56.67%) for *kbyk*.

3.4.4 - Drawback of the *k*-block splitting technique

The main target of the *k*-block splitting technique was to reduce the global communication time while keeping the good performance of the irregular partitioning technique. Experimental results have besides shown that our CGM-based parallel solutions based on the *k*-block splitting technique, among which the one using the *k*-block by *k*-block evaluation strategy was the best, are better than our solu-



(a) *Global communication time* **(b)** *Total execution time*

Figure 54 – *Global communication time and total execution time for* $n = 40960$*,* $p \in \{32,\ldots,128\}$*, and* $k \in \{0,\ldots,4\}$ *while solving the MPP with the k-block splitting technique using the k-block by k-block evaluation strategy*

tions based on the irregular partitioning technique and that they are scalable to the increase of the data size and the number of processors.

However, these solutions are not scalable to the increase of the number of fragmentations. In fact, the blocks of the first diagonals which are not fragmented (e.g. the first and second diagonal in Figure 48d) are split either into $2^{k-1}(2^k+1)$ k-blocks (for those belonging to the first diagonal), or into 4^k k-blocks. For example on thirty-two processors, after performing four fragmentations, the blocks of the first diagonal will be subdivided into $2^3(2^4+1) = 136$ k-blocks, and those of the second, third, and fourth diagonal will be divided into $4^4 = 256$ k-blocks. When $n = 40960$, each k-block will be a square matrix of size 320×320. However, the blocks on the first diagonals, and therefore the k-blocks that make them up, do not require a very high computational load compared to the blocks on the last diagonals. As a result, the evaluation of the k-blocks seen in the previous example will not take too much time and will lead to a communication overhead as a consequence of the huge amount of communication that will have to be done by the processors to exchange very small data in a short time. Recall that the communication overhead raises the latency time of processors, which accounts for most of the global communication time. Figures 54a and 54b respectively illustrate the consequences of this shortcoming on the global communication time and the total execution time when $n = 40960$. They show that the performance of our CGM-based parallel solution using the k-block by k-block evaluation strategy deteriorates from the third fragmentation.

To solve this drawback, one idea would be to fix the number of subblocks belonging to a large-size block so that this number no longer increases as the number of fragmentations rises to minimize the communication between processors and

consequently to avoid communication overhead. The partitioning strategy based on this approach should also be applied to solve the OBST problem because the k-block splitting technique is not suitable to solve it. Section 3.5 presents a strategy that meets these expectations.

3.5 - Third dynamic graph partitioning : four-splitting technique

To avoid the communication overhead caused by the k-block splitting technique while reducing the latency time of processors, the four-splitting technique consists of splitting the large-size blocks into four small-size blocks (or subblocks) after performing k fragmentations. The subblocks of the blocks belonging to the lth level of fragmentation must have the same size as the blocks belonging to the $(l+1)$th level of fragmentation. The goal is the same as the k-block splitting technique, that is, allows processors to start the evaluation of blocks as soon as possible. Hence, evaluating a block by a single processor will consist of computing and communicating each subblock contained in this block.

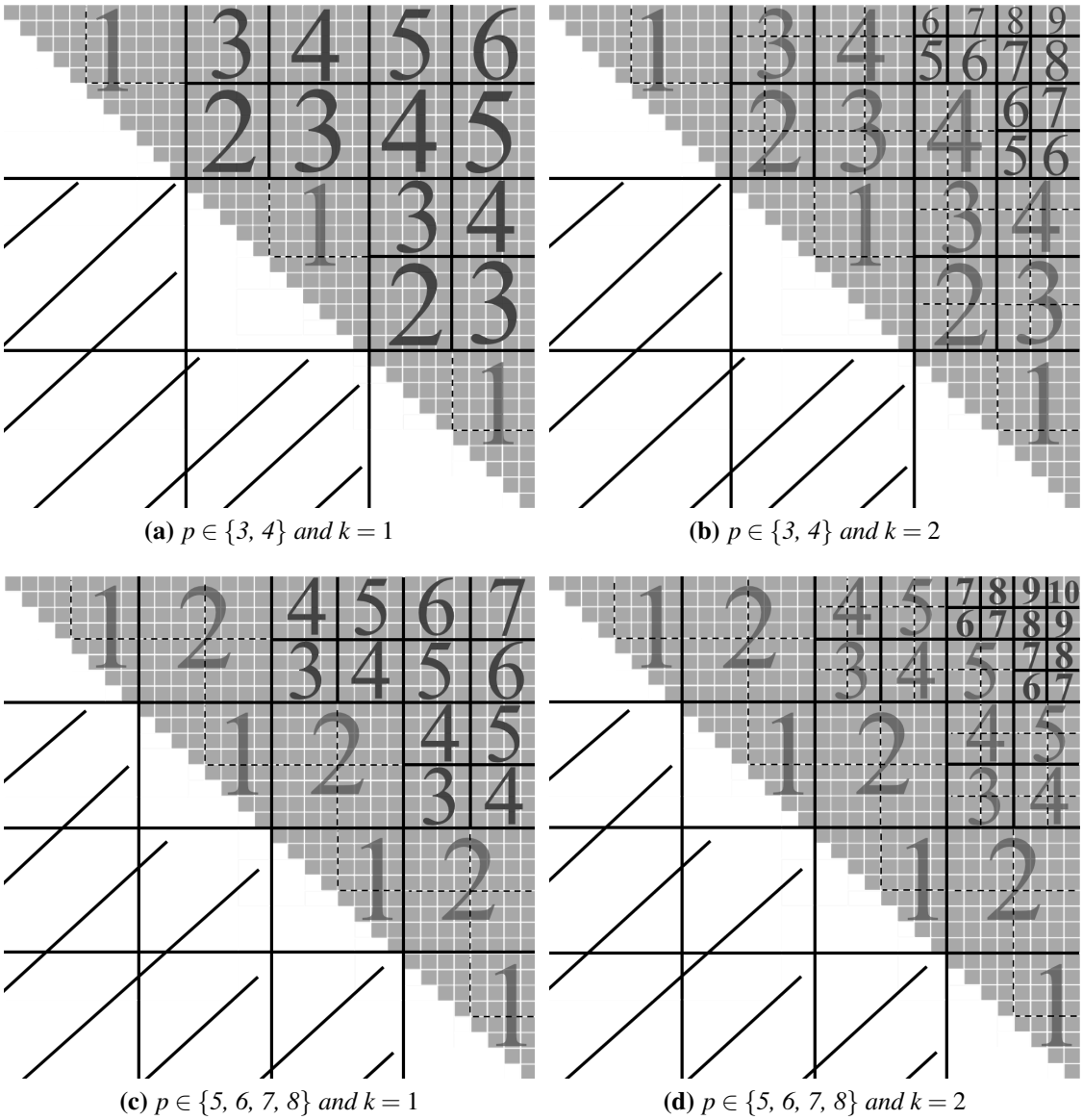
By denoting $f(p) = \lceil \sqrt{2p} \rceil$, $\theta(n,p) = \left\lceil \frac{n}{f(p)} \right\rceil$, and $\theta(n,p,l) = \left\lceil \frac{\theta(n,p)}{2^l} \right\rceil$, formally, we subdivide the shortest path matrix SP into blocks (denoted by $SM(i,j)$), and split the large-size blocks into four subblocks. Thus, a block $SM(i,j)$ belonging to the lth level of fragmentation, such that $l < k$, is a $\theta(n,p,l) \times \theta(n,p,l)$ matrix and is subdivided into four subblocks of size $\theta(n,p,l+1) \times \theta(n,p,l+1)$. The blocks of the kth level of fragmentation are not splitting into four as these are the smallest blocks. Figures 55a, 55b, 55c, and 55d depict four scenarios of this partitioning for $n = 32$, $k \in \{1,2\}$, and $p \in \{3, 4, 5, 6, 7, 8\}$.

Remark 6 *Some relevant points can be noticed about this partitioning :*

1 - *the blocks of the first diagonal are $\theta(n,p) \times \theta(n,p)$ upper triangular matrices splitting into tree subblocks;*

2 - *when $k = 1$, this partitioning technique is identical to the k-block splitting technique; for example, Figures 55a and 55c are the same as Figures 48a and 48c respectively where $k = 1$;*

3 - *whatever the number of fragmentations performed, the size of subblocks (which are the parts of blocks belonging to the lth level of fragmentation) does not change compared to the k-block splitting technique; for example,*



(a) $p \in \{3, 4\}$ *and* $k = 1$ **(b)** $p \in \{3, 4\}$ *and* $k = 2$

(c) $p \in \{5, 6, 7, 8\}$ *and* $k = 1$ **(d)** $p \in \{5, 6, 7, 8\}$ *and* $k = 2$

Figure 55 – *Four-splitting technique of the shortest path matrix for* $n = 32$, $k \in \{1,2\}$, *and* $p \in \{3, 4, 5, 6, 7, 8\}$. *For* $p \in \{3, 4\}$, *SP is partitioned into fifteen blocks and twenty-one subblocks when* $k = 1$, *and into twenty-four blocks and fifty-seven subblocks when* $k = 2$. *For* $p \in \{5, 6, 7, 8\}$, *SP is partitioned into nineteen blocks and thirty-six subblocks when* $k = 1$, *and into twenty-eight blocks and seventy-two subblocks when* $k = 2$

the subblocks of the second diagonal are square matrix of size 4×4 *when* $k = 1$ *in Figure 48c, and are square matrix of size* 2×2 *in Figure 48c when* $k = 2$. *But in Figures 55c and 55d, the subblocks of the second diagonal are square matrix of size* 4×4.

Lemma 15 *After partitioning the shortest path matrix, the total number of subblocks is equal to:*

$$C = \frac{S(S-1)}{2} - 3\left(\Delta(S+1) + \left\lceil \frac{S}{2} \right\rceil\right) + 2k\left[(S+1)(S+2\Delta) - \left\lceil \frac{S}{2} \right\rceil\left(\left\lceil \frac{S}{2} \right\rceil - 1\right)\right]$$

with $S = f(p)$ *and* $\Delta = (S \bmod 2)$.

Proof. The proof of this lemma can be easily deduced from Lemma 12. The blocks of the first diagonal are splitting into three subblocks, and the others are splitting into four subblocks except those belonging to the kth level of fragmentation. By considering $\Delta = (S \bmod 2)$ the variable which determines the parity of S, the total number of subblocks is equal to :

$$\begin{aligned}
C &= 4(k-1) \times \left(\frac{S(S+1) - \left\lceil \frac{S}{2} \right\rceil\left(\left\lceil \frac{S}{2} \right\rceil - 1\right)}{2} + \Delta(S+1)\right) + \left\lceil \frac{S}{2} \right\rceil + \frac{S(S+1)}{2} \\
&+ \Delta(S+1) + 4\left(\frac{S(S+1) - \left\lceil \frac{S}{2} \right\rceil\left(\left\lceil \frac{S}{2} \right\rceil + 1\right)}{2} - S\right) + 3S \\
&= 4(k-1) \times \frac{(S+1)(S+2\Delta) - \left\lceil \frac{S}{2} \right\rceil\left(\left\lceil \frac{S}{2} \right\rceil - 1\right)}{2} + \Delta(S+1) + \frac{S(5S+3)}{2} \\
&- 2\left\lceil \frac{S}{2} \right\rceil^2 - \left\lceil \frac{S}{2} \right\rceil \\
&= \frac{S(S-1)}{2} - 3\left(\Delta(S+1) + \left\lceil \frac{S}{2} \right\rceil\right) + 2k\left[(S+1)(S+2\Delta) - \left\lceil \frac{S}{2} \right\rceil\left(\left\lceil \frac{S}{2} \right\rceil - 1\right)\right]
\end{aligned}$$

■

The choice to split blocks into four subblocks is not trivial. Indeed, it allows keeping a coherence with the first step of this partitioning, which is to apply the irregular partitioning technique. Moreover, the four-splitting technique ensures that processors do not communicate too many subblocks to avoid communication overhead. However, the k-block splitting technique starts the evaluation of blocks earlier than this technique since k-blocks are the smallest blocks. This could have a significant impact on the latency time of processors.

The analysis of dependencies between subblocks can be derived from the two previous partitioning techniques. Unlike the k-block splitting technique, which was not suitable for solving the OBST problem due to the speedup of Knuth (1971), the four-splitting technique can be used to solve this problem. Indeed, the subblocks can be progressively evaluated either while solving the MPP, or non-progressively

Algorithm 27 Our CGM-based parallel algorithm based on the four-splitting technique to solve the MPP

1: $maxDiag \leftarrow f(p) + k \times (\lceil f(p)/2 \rceil + 1)$;
2: **for** $u = 1$ to $maxDiag$ **do**
3: **for each** subblock ρ belonging in the block $SM(i, j)$ **do**
4: **Finalization** of the evaluation of the block $SM(i, j)$ of diagonal u : compute the shortest path costs to nodes of the subblock ρ;
5: **Communication** of the subblock ρ to the processors that detain upper blocks and right blocks;
6: **for each** $\rho' \in SM(i', j')$ of diagonals $(u+1, \ldots, \min\{2 \times (u-1), maxDiag\})$ **do**
7: **Update** the shortest path costs to nodes of the subblock ρ';

evaluated while solving the OBST problem. The corresponding CGM-based parallel algorithms are described in Sections 3.5.1 and 3.5.2 respectively.

3.5.1 - CGM-based parallel algorithm to solve the MPP

Our CGM-based parallel algorithm based on the four-splitting technique to solve the MPP is given by Algorithm 27. Since the k-block by k-block evaluation approach has shown the better performance than the diagonal by diagonal evaluation approach in Section 3.4.3, a similar evaluation approach is used in Algorithm 27. Indeed, after computing each subblock containing in a block, it is immediately communicated to processors that need these subblocks for updating (line 7 in Algorithm 27) or for finalizing (line 4 in Algorithm 27) the computations of values in next steps.

Theorem 17 *Our CGM-based parallel solution based on the four-splitting technique to solve the MPP runs in $O\left(n^3/p\right)$ execution time with $O\left(k\sqrt{p}\right)$ communication rounds in the worst case.*

Proof. Let $S = f(p) = \lceil \sqrt{2p} \rceil$ and $\beta = (S \bmod 2)$. A single processor computes and communicates:

- three subblocks at the first diagonal of blocks;
- four subblocks from the diagonal of blocks 2 to $\lfloor S/2 \rfloor - 1$;
- four subblocks for each $(\lceil S/2 \rceil + 1)$ diagonals of blocks belonging to the lth level of fragmentation such that $1 \leq l < k$;
- one subblock for each $(S + \beta)$ diagonals of blocks belonging to the kth level of fragmentation.

During the computation rounds, evaluating each subblock of a block belonging to the lth level of fragmentation requires $O\left(\frac{n^3}{2^{3(l+1)}\times(2p)^{3/2}}\right) = O\left(\frac{n^3}{8^{l+1}\times p\sqrt{p}}\right)$ local computation time. So, the evaluation of each diagonal of blocks required :

$$D = 3\times O\left(\frac{n^3}{8\times p\sqrt{p}}\right) + 4\left(\left\lfloor\frac{S}{2}\right\rfloor - 1\right)\times O\left(\frac{n^3}{8\times p\sqrt{p}}\right) + 4\left(\left\lceil\frac{S}{2}\right\rceil + 1\right)\times$$
$$O\left(\frac{n^3}{8^2\times p\sqrt{p}}\right) + 4\left(\left\lceil\frac{S}{2}\right\rceil + 1\right)\times O\left(\frac{n^3}{8^3\times p\sqrt{p}}\right) + \cdots + 4\left(\left\lceil\frac{S}{2}\right\rceil + 1\right)\times$$
$$O\left(\frac{n^3}{8^k\times p\sqrt{p}}\right) + \left(\left\lceil\frac{S}{2}\right\rceil + 1\right)\times O\left(\frac{n^3}{8^k\times p\sqrt{p}}\right) + (S+\beta)\times O\left(\frac{n^3}{8^k\times p\sqrt{p}}\right)$$
$$= O\left(\frac{n^3}{p}\right)$$

The number of communication rounds is equal to :

$$E = 3 + 4\left(\left\lfloor\frac{S}{2}\right\rfloor - 1\right) + 4(k-1)\left(\left\lceil\frac{S}{2}\right\rceil + 1\right) + S + \beta = O\left(k\sqrt{p}\right)$$

Therefore, this algorithm requires $O\left(n^3/p\right)$ execution time with $O\left(k\sqrt{p}\right)$ communication rounds. ■

3.5.2 - CGM-based parallel algorithm to solve the OBST problem

The four-splitting technique can be adapted to evaluate the subblocks that are part of a block in a non-progressive fashion compared to the k-block splitting technique. Recall that when the latter is used, the number of subblocks (or k-blocks) grows drastically as the number of fragmentations rises. When a subblock is non-progressively evaluated by a processor, it must receive all the data of subblocks that it needs to start computations. However, since the subblocks are numerous and small, computing and communicating them to processors that need them will lead to communication overhead (see Section 3.4.4). With the four-splitting technique, the blocks are subdivided into at most four regardless of the number of fragmentations performed. This reduces the number of communication rounds and the number of steps to evaluate a whole block (it will be done in at most four steps).

Denote the four subblocks of given block by *LL*, *LU*, *RL*, and *RU*, which respectively correspond to the subblock located in the leftmost lower corner, the leftmost upper corner, the rightmost lower corner, and the rightmost upper corner. Figures 56a, 56b, 56c, 56d, 56e, and 56f illustrate the different computation and communication steps of these subblocks by a processor :

- At step 0 in Figure 56a, no subblocks have started to be evaluated. This means that the processor is still receiving data of subblocks on which *LL* depends.

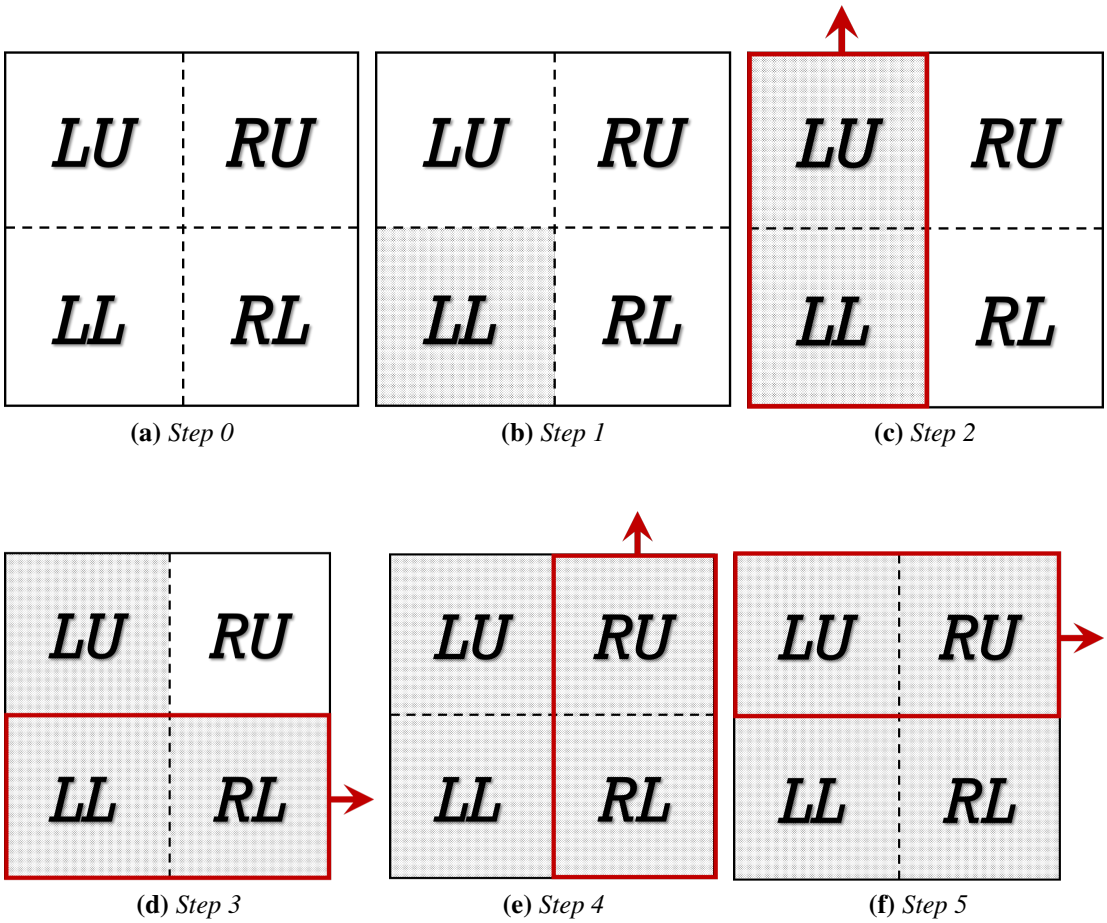
Algorithm 28 Our CGM-based parallel algorithm based on the four-splitting technique to solve the OBST problem

1: **for** $d = 1$ to $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ **do**
2: **Computation** of the subblocks *LL* and *LU* of blocks belonging to the round d using Algorithm 11;
3: **Communication** of entries (*Tree* and *Cut* tables) of *LL* and *LU* required for computing each block of rounds $\{d+1, d+2, \ldots, f(p) + k \times (\lceil f(p)/2 \rceil + 1)\}$;
4: **Computation** of the subblock *RL* using Algorithm 11;
5: **Communication** of entries of *LL* and *RL*;
6: **Computation** of the subblock *RU* using Algorithm 11;
7: **Communication** of entries of *LU*, *RL*, and *RU*;

- At step 1 in Figure 56b, *LL* is computed but it is not communicated because a communication is not necessary at this step. Indeed, the processors which will receive *LL* will not be able to start the computation as soon as possible because the evaluation is done in a non-progressive fashion. The ideal would be to wait to compute *LU* before communicating them together.
- At step 2 in Figure 56c, the processor starts by receiving the lower subblocks that are in the same row as *LU*. Then, it computes *LU*. Finally, it communicates *LL* and *LU* to processors that need them. For example, the processor P_5 which evaluates the block of the second diagonal in Figure 42 will send *LL* and *LU* to P_1 and P_3.
- At step 3 in Figure 56d, the processor receives the lower subblocks that are in the same column as *RL*. Then, it computes *RL*. Finally, it communicates *LL* and *RL* to processors that need them. By taking the previous example, P_5 will send *LL* and *RL* to P_0 and P_2.
- At step 4 in Figure 56e, the processor computes *RU*. Thereafter, it communicates *RL* and *RU* to processors that need them. By taking the previous example, P_5 will send *RL* and *RU* to P_1 and P_4.
- At step 5 in Figure 56f, all subblocks have been computed. The processor communicates *LU* and *RU* to processors that need them. By taking the previous example, P_5 will send *LU* and *RU* to P_3.

Our CGM-based parallel algorithm based on the four-splitting technique to solve the OBST problem is given by Algorithm 28.

Theorem 18 *Our CGM-based parallel solution based on the four-splitting technique requires* $O\left(n^2/\sqrt{p}\right)$ *execution time with* $O\left(k\sqrt{p}\right)$ *communication rounds in the worst case to solve the OBST problem.*



(a) *Step 0* **(b)** *Step 1* **(c)** *Step 2*

(d) *Step 3* **(e)** *Step 4* **(f)** *Step 5*

Figure 56 – *Steps to evaluate the four subblocks of a given block while solving the OBST problem with the four-splitting technique. When a subblock is computed and needs to be communicated, it is colored in gray*

Proof. It is deducible from Theorem 17. ■

3.5.3 - Experimental results

This section highlights the results obtained from experimentations of our CGM-based parallel solutions based on the four-splitting technique to solve the MPP and the OBST problem. These experimentations have been performed on the MatriCS platform (described in Section 2.4.7) and on our Raspberry Pi cluster (described hereunder). The execution time, the speedup, and the efficiency are recorded in Tables 10, 11, 12, 13, and 14. Figures 57a, 57b, 57c, 57d, 58a, 58b, 58c, 58d, 58e, 58f, 58g, 58h, 59a, 59b, 60a, 60b, 60c, 60d, 61a, and 61b have been drawn from these tables. Our solutions based on the irregular partitioning technique, the k-block by k-block evaluation strategy, and the four-splitting technique are denoted by *frag*, *kbyk*, and *4s* respectively. Recall that *kbyk* and *4s* are similar when $k = 1$.

Table 10 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 40960\}$, $p = 32$, and $k \in \{2,3,4\}$ while solving the MPP with the four-splitting technique on the MatriCS platform*

	Total execution time			Speedup			Efficiency		
n	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$
4096	13.65	14.63	14.66	15.59	14.55	14.52	48.73	45.46	45.37
8192	107.31	115.40	119.74	20.86	19.40	18.69	65.19	60.62	58.42
12288	404.33	396.91	398.45	19.91	20.29	20.21	62.23	63.39	63.15
16384	1075.51	921.47	976.78	18.06	21.08	19.89	56.44	65.87	62.14
20480	2133.58	2005.46	2036.95	17.78	18.91	18.62	55.56	59.11	58.19
24576	3854.41	3799.59	3842.85	17.01	17.25	17.06	53.15	53.92	53.31
28672	6547.45	6299.63	6474.68	15.99	16.62	16.17	49.97	51.93	50.53
32768	10578.86	9890.23	10397.02	16.34	17.48	16.62	51.06	54.61	51.95
36864	15842.78	14543.44	15667.99	15.39	16.76	15.56	48.08	52.38	48.62
40960	23670.69	19311.50	20826.53	16.22	19.88	18.44	50.69	62.14	57.62

Table 11 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{2,3,4\}$ while solving the MPP with the four-splitting technique on the MatriCS platform*

	Total execution time			Speedup			Efficiency		
p	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$
$n = 36864$									
64	6967.70	6917.32	6994.31	34.99	35.24	34.85	54.67	55.06	54.46
96	4981.26	4915.05	4932.40	48.94	49.60	49.42	50.98	51.66	51.48
128	4083.83	3803.69	3873.38	59.69	64.09	62.94	46.63	50.07	49.17
$n = 40960$									
64	9926.20	9758.72	9874.92	38.68	39.35	38.88	60.44	61.48	60.76
96	7439.89	7245.45	7296.02	51.61	53.00	52.63	53.76	55.20	54.82
128	5635.09	5449.50	5540.51	68.14	70.46	69.30	53.23	55.05	54.14

Table 12 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 40960\}$, $p = 32$, and $k \in \{1, \ldots, 5\}$ while solving the OBST problem with the four-splitting technique on the MatriCS platform*

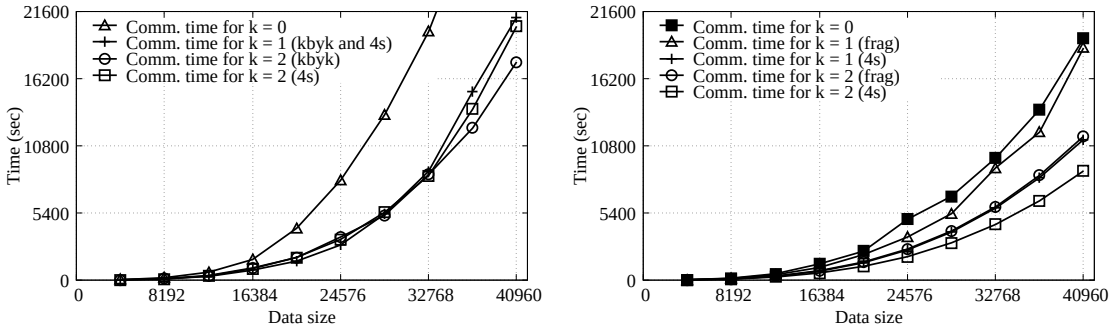
	Total execution time					Speedup					Efficiency				
n	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
4096	11.50	9.90	8.94	8.58	8.52	5.42	6.29	6.97	7.27	7.32	16.94	19.67	21.79	22.71	22.87
8192	97.34	77.30	70.76	69.10	68.64	5.28	6.65	7.27	7.44	7.49	16.51	20.79	22.71	23.26	23.42
12288	338.27	266.48	243.43	236.72	235.48	5.27	6.70	7.33	7.54	7.58	16.48	20.92	22.90	23.55	23.68
16384	813.47	634.41	585.65	572.21	568.37	5.30	6.80	7.37	7.54	7.59	16.58	21.26	23.03	23.57	23.73
20480	1605.26	1244.97	1147.37	1123.66	1116.53	5.32	6.87	7.45	7.61	7.65	16.64	21.45	23.28	23.77	23.92
24576	2793.95	2151.68	1977.78	1937.93	1930.08	5.33	6.92	7.53	7.69	7.72	16.67	21.64	23.54	24.03	24.12
28672	4458.84	3444.42	3155.25	3089.17	3076.31	5.34	6.91	7.54	7.71	7.74	16.68	21.60	23.57	24.08	24.18
32768	6682.07	5153.27	4718.72	4616.36	4587.60	4.98	6.45	7.05	7.21	7.25	15.56	20.17	22.03	22.52	22.66
36864	9578.16	7353.35	6729.28	6572.38	6529.41	5.05	6.58	7.19	7.36	7.41	15.78	20.55	22.46	22.99	23.14
40960	13219.62	10120.25	9252.37	9013.49	8961.54	5.27	6.89	7.54	7.74	7.78	16.48	21.53	23.55	24.17	24.31

Table 13 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{1, \ldots, 5\}$ while solving the OBST problem with the four-splitting technique on the MatriCS platform*

	Total execution time					Speedup					Efficiency				
p	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$n = 36864$															
64	6849.69	5268.74	4744.92	4609.23	4492.70	7.06	9.18	10.19	10.49	10.76	11.03	14.34	15.92	16.39	16.82
96	5811.06	4336.53	3960.81	3863.48	3846.97	8.32	11.15	12.21	12.52	12.57	8.67	11.62	12.72	13.04	13.09
128	5166.80	3858.00	3488.67	3407.10	3391.69	9.36	12.53	13.86	14.19	14.26	7.31	9.79	10.83	11.09	11.14
$n = 40960$															
64	9564.80	7256.84	6333.78	6208.79	6177.57	7.29	9.61	11.01	11.23	11.29	11.39	15.01	17.20	17.55	17.63
96	7980.85	5954.35	5415.29	5299.00	5276.19	8.74	11.71	12.88	13.16	13.21	9.10	12.20	13.41	13.71	13.77
128	7115.22	5316.16	4815.67	4688.91	4670.79	9.80	13.12	14.48	14.87	14.93	7.66	10.25	11.31	11.62	11.66

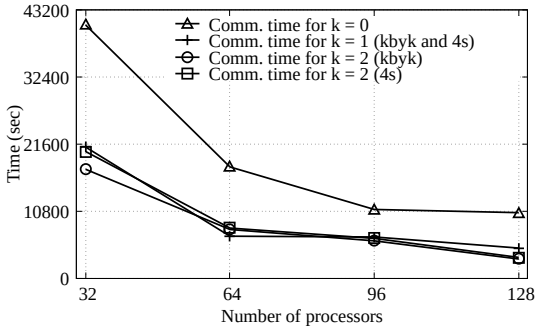
Table 14 – *Total execution time (in seconds), speedup, and efficiency (in %) for $n \in \{4096, \ldots, 16384\}$, $p \in \{1, 32\}$, and $k \in \{1, \ldots, 5\}$ while solving the OBST problem with the four-splitting technique on our Raspberry Pi cluster*

	Total execution time						Speedup					Efficiency				
n	$p=1$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
4096	152.45	70.53	61.74	50.45	51.68	52.24	2.16	2.47	3.02	2.95	2.92	6.75	7.72	9.44	9.22	9.12
8192	1252.78	553.87	425.86	415.23	385.80	391.02	2.26	2.94	3.02	3.25	3.20	7.07	9.19	9.43	10.15	10.01
12288	4384.93	2670.32	1615.90	1448.08	1318.49	1351.73	1.64	2.71	3.03	3.33	3.24	5.13	8.48	9.46	10.39	10.14
16384	10660.09	4856.32	4034.42	3870.51	3610.49	3464.09	2.20	2.64	2.75	2.95	3.08	6.86	8.26	8.61	9.23	9.62

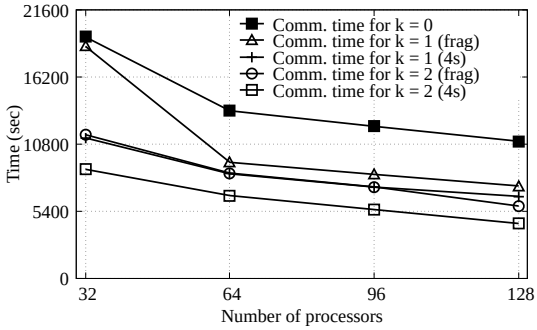


(a) *MPP for* $p = 32$

(b) *OBST problem for* $p = 32$



(c) *MPP for* $n = 40960$



(d) *OBST problem for* $n = 40960$

Figure 57 – *Global communication time for* $n \in \{4096, \ldots, 40960\}$, $p \in \{32, \ldots, 128\}$, *and* $k \in \{0, 1, 2\}$ *while solving the MPP and the OBST problem with the four-splitting technique on the MatriCS platform*

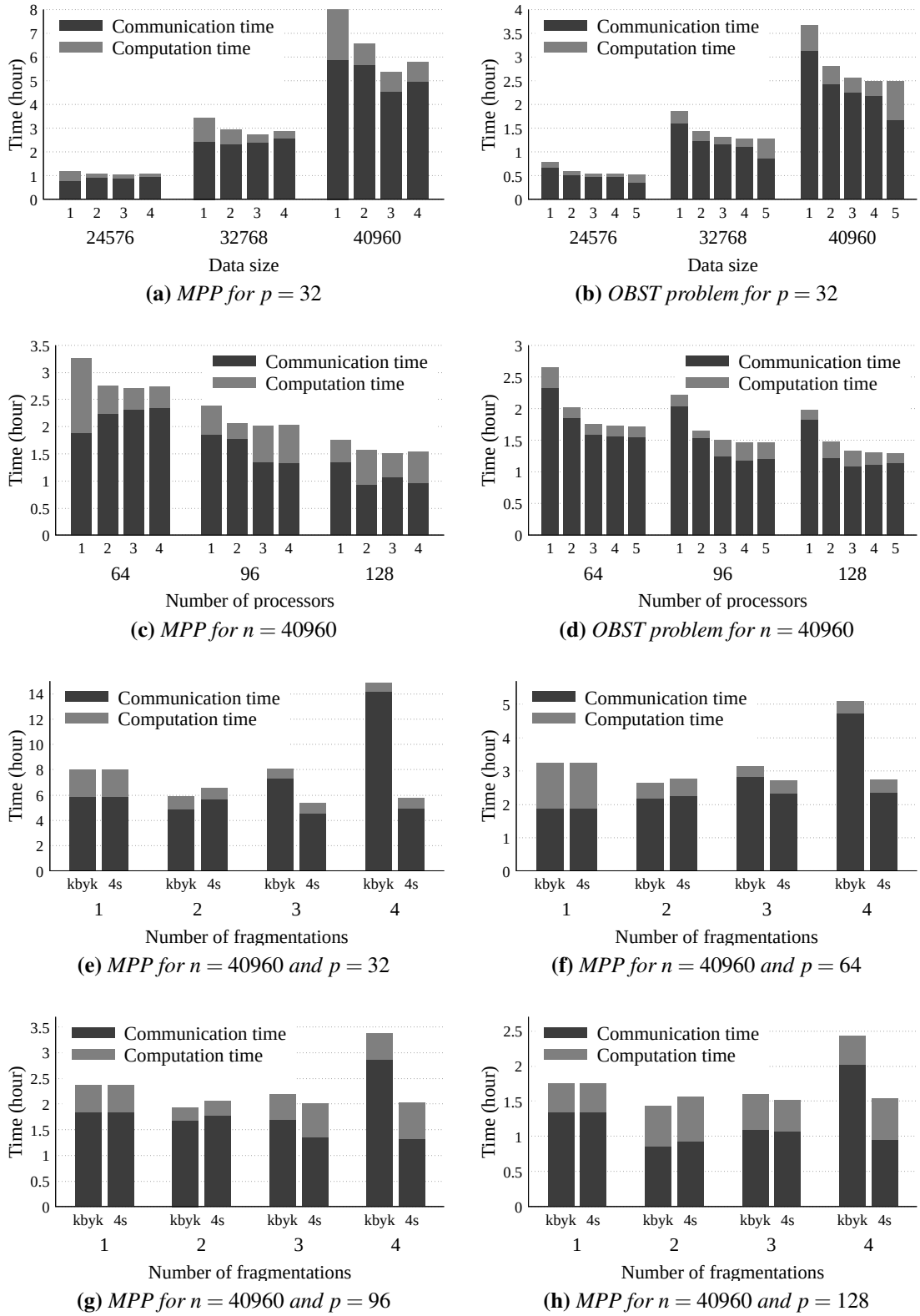
Evolution of the global communication time

Figures 57a and 57c compare the global communication time of *kbyk* and *4s* while solving the MPP by performing one and two fragmentations. Figure 57a shows that from $n = 4096$ to 24576, the global communication time of *kbyk* and *4s* is better when performing one fragmentation than when performing two. Indeed, while solving the MPP, the evaluation of the small subblocks requires less time than the large ones; and thus requires less latency time of processors. Therefore, decreasing the size of these subblocks by performing more fragmentations will lead to increase the global communication time and minimize the overall computation time. This will lead to a better total execution time when performing more than one fragmentation. For example, on thirty-two processors when $n = 24576$ in Figure 58a, the total execution time is made up of 34.39% of computation time and 65.61% of communication time when $k = 1$, and 15.30% of computation time and 84.70% of communication time when $k = 2$. Now when comparing the global communication time of *kbyk* and *4s*, it can be noticed that *4s* is better than *kbyk*. Indeed, it is necessary to split the blocks into at most four subblocks because their sizes are not large enough. Excessive communications degrading the global communication time can

be achieved when there are more subblocks than necessary. Figure 57a also shows that from $n = 28800$, the global communication time of *kbyk* when $k = 2$ is better than that of *4s* when $k = 1$ and $k = 2$. Indeed, large subblocks require more time to be evaluated; and thus, require more than one fragmentation to minimize the latency time of processors (this observation is not true in all cases because, for example on sixty-four processors in Figure 57c, the global communication time of *kbyk* and *4s* when $k = 1$ is smaller than when $k = 2$). In addition, it is necessary to split large-size blocks into more than four subblocks to allow processors to start or continue evaluating their blocks as soon as possible. For example on thirty-two processors when $n = 40960$, *kbyk* decreases the global communication time on average by 56.97% and *4s* decreases it on average by 49.96% when $k = 2$. However, Figure 58a shows that *4s* decreases the global communication time on average by 59.87% when $k = 3$. This is due to the fact that blocks belonging to the last level of fragmentation are quite large and require an additional fragmentation when $k = 2$. In contrast, Figures 58f, 58g, and 58h show that from sixty-four to one hundred and twenty-eight processors, the global communication time of *kbyk* when $k = 2$ is better than that of *4s* when $k = 3$. This is because increasing the number of processors results in decreasing the size of blocks (as well as subblocks); therefore, applying a third fragmentation results in minimizing the global communication time, which is not enough to be better than *kbyk* when $k = 2$. Nevertheless, it would have been wise to split blocks into more than four subblocks when $k = 3$ to decrease the latency time as much as possible.

Figures 57b and 57d show that the global communication time of *4s* is smaller than *frag* while solving the OBST problem by performing one and two fragmentations. It significantly decreases when the number of fragmentations increases. For example, on thirty-two processors when $n = 40960$, the global communication time of *frag* (respectively *4s*) decreases on average by 3.87% (respectively 41.84%) when $k = 1$ and on average by 40.49% (respectively 54.83%) when $k = 2$. These results were expected because *4s* minimizes the latency time of processors by allowing them to start evaluating subblocks as soon as the data they need are available. Grouping the subblocks into two before communicating also contributes to this minimization because it limits the number of messages to be exchanged in the network.

Figures 58a and 58c (respectively 58b and 58d) show that in general the global communication time and the overall computation time of *4s* gradually decrease while solving the MPP (respectively the OBST problem) by successively performing four (respectively five) fragmentations. Nevertheless, Figures 58a and 58c



(a) *MPP for* $p = 32$ **(b)** *OBST problem for* $p = 32$

(c) *MPP for* $n = 40960$ **(d)** *OBST problem for* $n = 40960$

(e) *MPP for* $n = 40960$ *and* $p = 32$ **(f)** *MPP for* $n = 40960$ *and* $p = 64$

(g) *MPP for* $n = 40960$ *and* $p = 96$ **(h)** *MPP for* $n = 40960$ *and* $p = 128$

Figure 58 – *Comparison of the overall computation time and the global communication time for* $n \in \{24576, 32768, 40960\}$*,* $p \in \{32, \ldots, 128\}$*, and* $k \in \{1, \ldots, 5\}$ *while solving the MPP and the OBST problem with the four-splitting technique on the MatriCS platform*

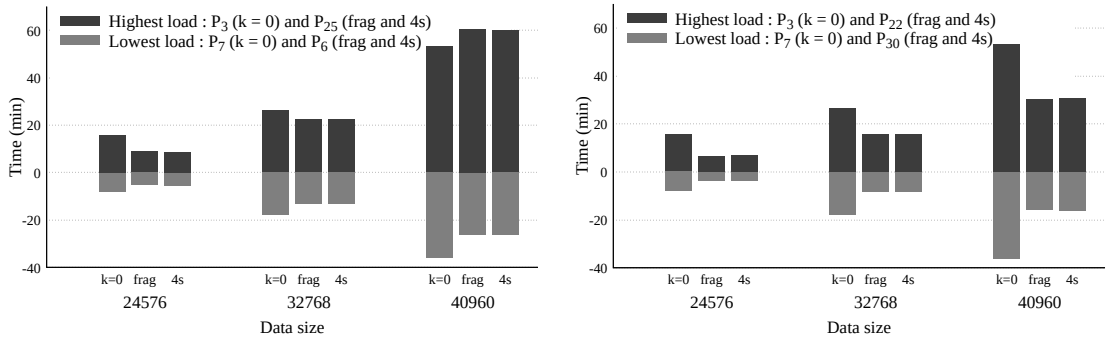
show that the global communication time of *4s* degrades from the fourth fragmentation because on thirty-two processors when $n = 40960$, it decreases in average by 48.17% when $k = 1$, in average by 49.96% when $k = 2$, in average by 59.87% when $k = 3$, and in average by 56.12% when $k = 4$. This is due to the fact that from a certain number of fragmentations, the small subblocks which are close to the optimal solution (and thus require a high evaluation time using the sequential algorithm of Godbole (1973)) do not need to be fragmented any more because they can lead to excessive communications; and thus the latency time of processors will be increased instead of being reduced. This observation is not the same for the OBST problem as shown in Figures 58b and 58d because until the fifth fragmentation the global communication time does not deteriorate. For example, on thirty-two processors when $n = 40960$, it decreases in average by 41.84% when $k = 1$, in average by 54.83% when $k = 2$, in average by 58.18% when $k = 3$, in average by 59.43% when $k = 4$, and in average by 69.08% when $k = 5$. This is due to the sequential algorithm of Knuth (1971), which is used to evaluate small subblocks that are close to the optimal solution. Since they do not require a high evaluation time, processors that need these subblocks will not have to wait long to receive them.

Evolution of the load-balancing of processors

Figures 59a and 59b show that while solving the OBST problem, our CGM-based parallel solution based on the four-splitting technique (*4s*) kept the good performance of our CGM-based parallel solution based on the irregular partitioning technique (*frag*) with respect to the load-balancing of processors since they promote the load balancing when the number of fragmentations increases. Indeed, the load-balancing of processors for *frag* and *4s* are nearly the same. When $n = 40960$ and $k = 1$, the lowest load narrows down to 27.52% for *frag* and to 27.24% for *4s*; and the highest load increases up to 11.89% for *frag* and to 11.60% for *4s*. In the same vein when $k = 2$, the lowest load and the highest load decrease on average by 56.63% and 42.98% for *frag*, and on average by 55.73% and 42.77% for *4s*. As for the MPP, these results were predictable because *4s* (which is a special case of *kbyk*) are based on the irregular partitioning technique and snake-like mapping.

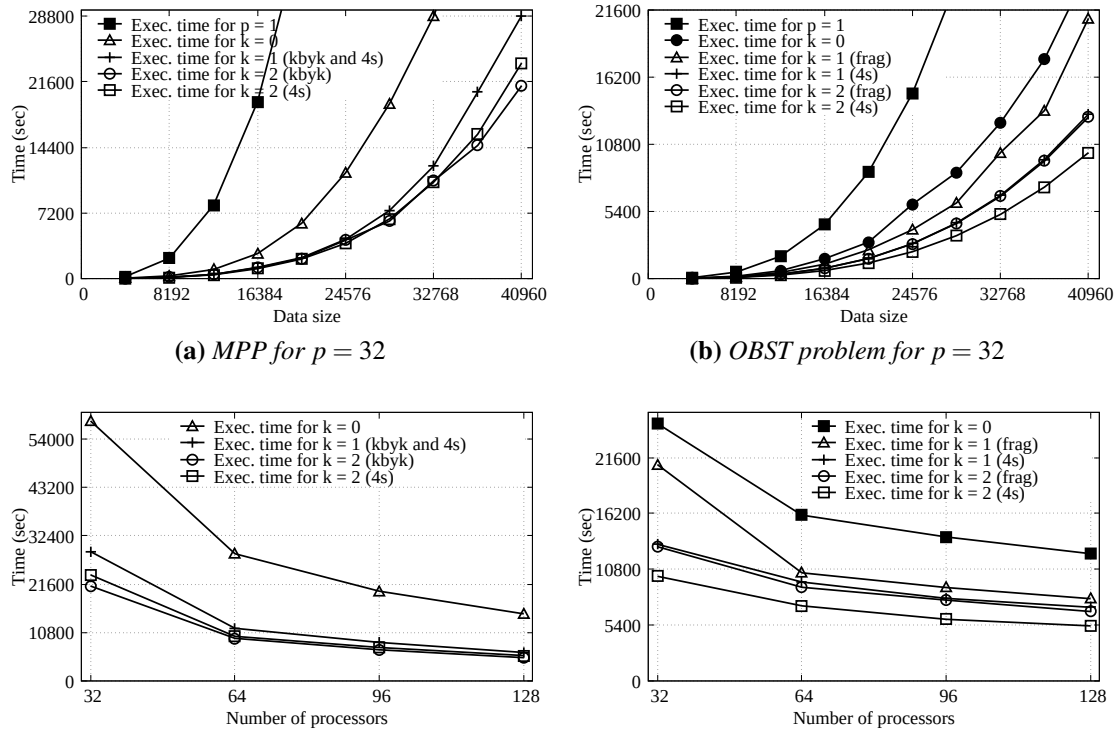
Evolution of the total execution time

Figures 58a, 58b, 58c, 58d, 58e, 58f, 58g, 58h, 60a, 60b, 60c, and 60d illustrate the total execution time of *frag*, *kbyk*, and *4s* while solving the MPP and the OBST problem by performing one, two, three, four, and five fragmentations. As observed in Sections 3.3.5 and 3.4.3, the global communication time (see above) has a huge



(a) $k = 1$ **(b)** $k = 2$

Figure 59 – *Load imbalance of processors for* $n \in \{24576, 32768, 40960\}$*,* $p = 32$*, and* $k \in \{0,1,2\}$ *while solving the OBST problem with the four-splitting technique on the MatriCS platform*



(a) *MPP for* $p = 32$ **(b)** *OBST problem for* $p = 32$

(c) *MPP for* $n = 40960$ **(d)** *OBST problem for* $n = 40960$

Figure 60 – *Total execution time for* $n \in \{4096, \ldots, 40960\}$*,* $p \in \{32, \ldots, 128\}$*, and* $k \in \{0,1,2\}$ *while solving the MPP and the OBST problem with the four-splitting technique on the MatriCS platform*

impact on the total execution time, and consequently on the speedup and the efficiency (as shown in Tables 10, 11, 12, and 13). The results are obvious while solving the MPP:

- From $n = 4096$ to 24576 on thirty-two processors, the total execution time of *4s* is better than *kbyk* when performing two fragmentations. For example

when $n = 24576$ and $k = 2$, the speedup and the efficiency are equal to 15.59 and 48.71% for *kbyk*, and increase up to 17.01 and 53.15% for *4s*.

- From $n = 28800$ on thirty-two processors, the total execution time of *kbyk* is better than that of *4s* when performing two fragmentations. For example when $n = 40960$ and $k = 2$, the speedup and the efficiency are equal to 18.13 and 56.67% for *kbyk*, and narrow down to 16.22 and 50.69% for *4s*.

- From $n = 4096$ to 40960 on thirty-two processors, the total execution time of *4s* when performing three fragmentations is better than that of *kbyk* when performing two fragmentations. For example when $n = 40960$ and $k = 3$, the speedup and the efficiency are equal to 19.88 and 62.14% for *4s*.

- When $n = 36864$ and $n = 40960$ on sixty-four to one hundred and twenty-eight processors, the total execution time of *kbyk* when performing two fragmentations is better than *4s* when performing three fragmentations. For example on one hundred and twenty-eight processors when $n = 40960$, the speedup and the efficiency are equal to 74.36 and 58.09% for *kbyk* when $k = 2$, and narrow down to 70.46 and 55.05% for *4s* when $k = 3$.

- From $n = 4096$ to 40960 on thirty-two to one hundred and twenty-eight processors, the total execution time of *4s* degrades from the fourth fragmentation. For example when $n = 40960$ and $k = 4$, the speedup and the efficiency of *4s* are equal to 18.44 and 57.62% on thirty-two processors, and equal to 69.30 and 54.14% for *4s* on one hundred and twenty-eight processors.

In a nutshell, there is no better choice between *kbyk* and *4s* to solve the MPP because in some conditions *4s* is better than *kbyk* and in other conditions it is the opposite. However, we recommend to use *4s* to solve this problem since compared to *kbyk*, *4s* minimizes communication between processors and its performance does not degrade abruptly when the number of fragmentations increases.

Concerning the OBST problem, the following results can be noticed :

- From $n = 4096$ to 40960 on thirty-two to one hundred and twenty-eight processors, the total execution time of *4s* is better than that of *frag* when performing one and two fragmentations. For example, on thirty-two processors when $n = 40960$ and $k = 1$ (respectively $k = 2$), the speedup and the efficiency are equal to 3.33 and 10.41% (respectively 5.37 and 16.78%) for *frag*, and increase up to 5.27 and 16.48% (respectively 6.89 and 21.53%) for *4s*.

- When performing three, four, and five additional fragmentations, the total execution time of *4s* keeps good performance, although from the fourth fragmentation the performance gain is not meaningful anymore. For example, on thirty-two processors when $n = 40960$, the speedup and the efficiency are equal to 7.54 and 23.55% when $k = 3$, and increase up to 7.74 and 24.17% when $k = 4$, up to 7.78 and 24.31% when $k = 5$.

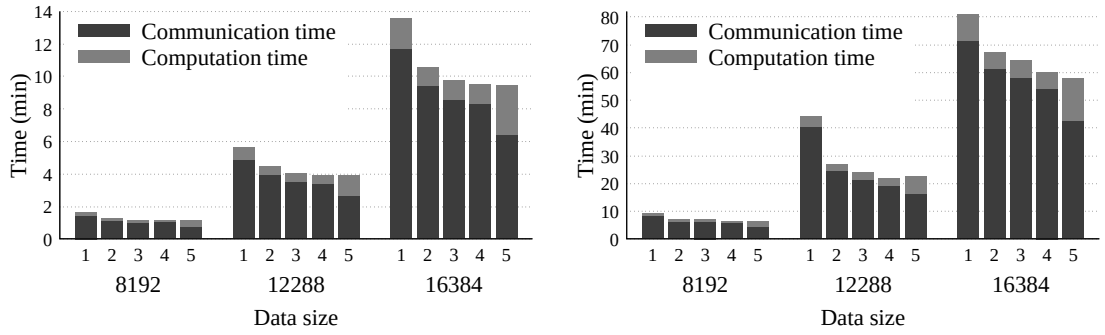
From all this, we can deduce that it is better to use our CGM-based parallel solution based on the four-splitting technique to solve the OBST problem since it outperforms the one using the irregular partitioning technique and it is scalable as the data size, the number of processors, and the number of fragmentations rise.

What is the ideal number of fragmentations ?

This is a legitimate question because as shown earlier, on the MatriCS platform the total execution time of *4s* degrades from the fourth fragmentation while solving the MPP but keeps good performance until the fifth fragmentation while solving the OBST problem. One idea to determine the ideal number of fragmentations would be to run our CGM-based parallel solution based on the four-splitting technique to solve the OBST problem for example on another cluster. If we obtain the same result that was archived on the MatriCS platform then this could guide us to determine this number. Otherwise, we will deduce that the ideal number of fragmentations depends on the platform where the parallel algorithm is executed.

To achieve this goal, we have built a cluster consisting of four compute nodes. Each node is a Raspberry Pi 4 computer model B composed of a 64-bit quad-core Cortex-A72 processor with 8GB of RAM and an operating system Raspbian GNU/Linux 11. All nodes are interconnected with Cat8 Ethernet cable through a TP-Link router (model TL-SG108E) providing 16Gbps throughput and the inter-processor communication has been ensured by OpenMPI version 4.1.2.

Figures 61a and 61b compare the overall computation time and global communication time of *4s* on thirty-two processors while solving the OBST problem by performing one, two, three, four, and five fragmentations on the MatriCS platform and on our Raspberry Pi cluster. It is obvious to notice that whatever the cluster, *4s* progressively reduces the global communication time when the number of fragmentations rises. However on the Raspberry Pi cluster, the performance of *4s* degrades when adding the overall computation time at the fifth fragmentation for $n = 8192$ and $n = 12288$ because the global communication time did not reduce enough to keep the good performance. For example when $n = 12288$ in Table 14, the speedup is equal to 1.64 when $k = 1$, and increases up to 2.71 when $k = 2$,



(a) *MatriCS platform* **(b)** *Raspberry Pi cluster*

Figure 61 – *Comparison of the overall computation time and the global communication time for* $n \in \{8192, 12288, 16384\}$*,* $p = 32$*, and* $k \in \{1, \ldots, 5\}$ *while solving the OBST problem with the four-splitting technique on on the MatriCS platform and our Raspberry Pi cluster*

up to 3.03 when $k = 3$, up to 3.33 when $k = 4$, but the speedup narrows down to 3.24 when $k = 5$. This result is due to the fact that the Raspberry Pi cluster does not have a communication network as fast as the MatriCS platform. Indeed, an additional fragmentation is no longer necessary from the fourth fragmentation. Thus, since the network is not very fast, excessive communication caused a drop in performance. In summary, we deduce that the ideal number of fragmentations depends on the architecture where the CGM-based parallel solution is executed.

Another relevant observation can be seen on the Raspberry Pi cluster in Figure 61b, where the total execution time of *4s* gradually decreases after five successive fragmentations when $n = 16384$. As shown in Table 14, the speedup is equal to 2.20 when $k = 1$, and increases up to 2.64 when $k = 2$, up to 2.75 when $k = 3$, up to 2.95 when $k = 4$, and up to 3.08 when $k = 5$. This is due to the fact that when the input data size increases, fragmentations are more and more required until the performance gain is not significant or drops. However, as seen earlier, the performance of *4s* degrades at the fifth fragmentation for $n = 8192$ and $n = 12288$. Consequently, we infer that the ideal number of fragmentations also depends on the input data size.

3.6 - Summary

This chapter presented CGM-based parallel solutions based on three dynamic graph partitioning techniques that reconcile the trade-off of minimizing the number of communication rounds and balancing the load of processors to solve the MPP and the OBST problem. First, we designed a dynamic graph model for the OBST prob-

lem, like in (Bradford, 1994) for the MPP, that can be used to solve this problem through a one-to-all shortest path algorithm. We improved the dynamic graph proposed by Bradford (1994) by removing some horizontal jumps in addition to the vertical jumps in the dynamic graph D'_n to avoid more redundant computations.

Then, we proposed an irregular partitioning technique of the dynamic graph to tackle the conflicting objectives. It consists in subdividing the dynamic graph into blocks of variable size. Our CGM-based parallel solutions using this technique require $O\left(n^3/p\right)$ execution time to solve the MPP and $O\left(n^2/\sqrt{p}\right)$ to solve the OBST problem, each with $O\left(k\sqrt{p}\right)$ communication rounds. Experimental results showed that these solutions minimize the overall computation time and the latency time, and balance better the load of processors than those using regular partitioning technique of the dynamic graph.

Finally, we proposed two techniques based on the irregular partitioning technique to reduce the latency time of processors by allowing them to start the evaluation of blocks as soon as possible. Indeed, varying the blocks' sizes does not enable processors to start evaluating small-size blocks as soon as the data they need are available, although these data are available before the end of the evaluation of large-size blocks. We first proposed the k-block splitting technique consisting in splitting the large-size blocks into a set of smaller-size blocks called *k-blocks*. Thus, evaluating a block by a single processor consists of computing and communicating each k-block contained in this block. Experimental results showed that our CGM-based parallel solution using this technique is better than previous but leads to communication overhead when k increases. So, we proposed the four-splitting technique consisting in splitting the large-size blocks into four small-size blocks. It avoids communication overhead and significantly reduces the latency time of processors. Experimental results are showed that our CGM-based parallel solutions using the four-splitting technique are scalable as the data size, the number of processors, and the number of fragmentations rise. These solutions are better than those using the irregular partitioning technique and the k-block splitting technique (although in some cases, the k-block splitting technique is better than the four-splitting technique while solving the MPP). We also conducted experiments to determine the ideal number of times the size of the blocks must be subdivided. The results showed that this number depends on the input data size and characteristics of the architecture of parallel computers where the solution is executed.

General Conclusion

CONTENTS

Re-stating the research problem

In this thesis, we have studied the parallelization on the coarse-grained multicomputer (CGM) model of a class of non-serial polyadic dynamic-programming problems that can be formulated by Equation (1.8). Especially, we are interested in the minimum cost parenthesizing problem (MPP), the matrix chain ordering problem (MCOP), the triangulation of a convex polygon (TCP) problem, and the optimal binary search tree (OBST) problem. The standard methodology for designing CGM-based parallel solutions to solve these problems is to partition the dependency graph into subgraphs (or blocks) of same size, then distribute these blocks fairly among processors, and finally compute them in a suitable evaluation order. Kengne et al. (2016) noticed that the execution time was related to the load balancing and the number of communication rounds. These criteria depend on the partitioning strategy and the distribution scheme strategy used when designing the CGM-based parallel solution. Kengne et al. (2016) showed that minimizing the number of communication rounds and balancing the load between processors are two conflicting objectives when the dependency graph (or the dynamic graph) is partitioned into blocks of the same size :

1 - When the dependency graph is subdivided into small-size blocks (Kechid and Myoupo, 2008a, 2008b, 2009), the load difference between processors

is small if one processor has one more block than another. However, the number of communication rounds will be high.

2 - When the dependency graph is subdivided into large-size blocks (Kengne and Myoupo, 2012; Myoupo and Kengne, 2014a, 2014b), the number of communication rounds is reduced since there are few blocks. However, the load of processors will be unbalanced.

By generalizing the ideas of the dependency graph partitioning and distribution scheme introduced in (Kechid and Myoupo, 2008a, 2009; Kengne and Myoupo, 2012; Myoupo and Kengne, 2014b), Kengne et al. (2016) proposed a CGM-based parallel solution that gives the end-user the choice to optimize one criterion according to their own goal.

The main drawback of this solution is the conflicting optimization criterion owing to the fact that the end-user cannot optimize more than one criterion. Moreover, these criteria have a significant impact on the latency time of processors, and consequently on the global communication time.

Results obtained and critical analysis

This thesis intended to design CGM-based parallel solutions that address the trade-off of minimizing the number of communication rounds and balancing the load of processors to solve the foregoing problems. To achieve our goal, we have proposed three techniques.

Irregular partitioning technique

This technique consists in subdividing the dependency graph into blocks of variable size. It ensures that the blocks of the first diagonals are of large sizes to minimize the number of communication rounds. Thereafter, it decreases these sizes along the diagonals to increase the number of blocks in these diagonals and allow processors to stay active longer. These blocks are distributed fairly among processors to minimize their idle time and balance the load between them. Our CGM-based parallel solutions using this technique require $O\left(n^3/p\right)$ execution time to solve the MPP and $O\left(n^2/\sqrt{p}\right)$ to solve the OBST problem, each with $O\left(k\sqrt{p}\right)$ communication rounds. n is the input data size, p is the number of processors, and k is the number of times the size of blocks is subdivided. The sequential algorithm of Godbole (1973) and the sequential algorithm of Knuth (1971) are respectively used in the local computation round while solving the MPP and the OBST problem.

Experimental results showed that these solutions balance the load of processors and minimize the overall computation time and the latency time. Consequently, our solutions significantly reduced the total execution time compared the regular partitioning technique proposed in (Kechid and Myoupo, 2008a, 2009; Kengne and Myoupo, 2012; Kengne et al., 2016; Myoupo and Kengne, 2014b).

Nevertheless, this technique also induces a high latency time of processors since it does not allow processors to start evaluating small-size blocks as soon as the data they need are available. Yet, these data are usually available before the end of the evaluation of large-size blocks.

k-block splitting technique

To reduce this latency time, the k-block splitting technique consists in splitting the large-size blocks into a set of smaller-size blocks called *k-blocks* after performing k fragmentations. Thus, to allow processors to start the evaluation of k-blocks as soon as possible, a single processor evaluates a block by computing and communicating each k-block contained in this block. This technique is essentially based on the progressive evaluation of blocks. Indeed, evaluating a block belonging to the diagonal d in a progressive fashion consists in starting at the diagonal $\lceil d/2 \rceil$. Therefore, this technique is suitable to solve the MPP because the sequential algorithm of Godbole (1973) gives the possibility to evaluate the nodes of blocks in this way. The sequential algorithm of Knuth (1971) in contrast does not allow performing this kind of evaluation to solve the OBST problem because it does not allow knowing when to start or continue the evaluation of a node. Thus, it was not meaningful and practical to apply the k-block splitting technique to solve the OBST problem because a lot of unnecessary computations could be performed and lead to poor performance.

For the MPP, depending on how processors evaluate the k-blocks contained in a block, we have derived two approaches :

1 - The diagonal by diagonal evaluation approach, which consists in evaluating a block by computing a set of k-blocks belonging to the same diagonal before communicating them. It runs in $O\left(n^3/4^k p\right)$ execution time with $O\left(2^k\sqrt{p}\right)$ communication rounds.

2 - The k-block by k-block evaluation approach, which consists in evaluating a block by computing and communicating each k-block contained in this block. It runs in $O\left(n^3/2^k p\right)$ execution time with $O\left(4^k\sqrt{p}\right)$ communication rounds.

Experimental results showed that these solutions are better than those based on the irregular partitioning technique since they significantly reduce the global communication time. They also showed that the second approach is better than the first because the second approach communicates a k-block as soon as a processor has finished computing it.

However, this technique induces a communication overhead when the number of fragmentations rises. In fact, since the k-block are numerous and small when k increases, a huge amount of communication must be done by processors to exchange data. Experimental results showed that this shortcoming deteriorates the performance of our CGM-based parallel solution as the communication overhead raises the latency time of processors, which accounts for most of the global communication time.

Four-splitting technique

The four-splitting technique avoids this communication overhead while reducing the latency time of processors by splitting the large-size blocks into four small-size blocks (or subblocks) after performing k fragmentations. Hence, evaluating a block by a single processor consists of computing and communicating each subblock contained in this block. This technique has been used to solve the MPP and the OBST problem since the subblocks can be evaluated either progressive or non-progressive fashion unlike the k-block splitting technique. Our CGM-based parallel solutions using this technique have the same complexity as those using the irregular partitioning technique.

Experimental results showed a good agreement with theoretical predictions. Our CGM-based parallel solutions using this technique were better than those using the irregular partitioning technique and the k-block splitting technique. These results also showed that while solving the MPP, when the input data size and the number of processors is large and a number of fragmentations is small, it is preferable to use k-block splitting technique than the four-splitting technique. We also conducted experimentations to determine the ideal number of times the size of the blocks must be subdivided. The results showed that this number depends on the input data size and the architecture where the solution is executed.

Our fast sequential algorithm for the MCOP and the TCP problem

CGM-based parallel solutions proposed by Kechid and Myoupo (2008b) and Myoupo and Kengne (2014a) to solve the MCOP and the TCP problem, where there is a one-to-one correspondence between them, exhibit the same trade-off addressed in

this thesis. Recall that these solutions are based on the $O\left(n^2\right)$-time sequential algorithm of Yao (1982), which change the form of the dependency graph by reducing the number of subproblems to be performed. We could easily propose CGM-based parallel solutions using the previous techniques. Nevertheless, after noticing that some unnecessary computations were performed by the sequential algorithm of Yao (1982) because the subproblems were evaluated according to their precedence order, we have begun by solving this issue. We have proposed a fast sequential algorithm consisting in organizing the evaluation of the subproblems according to their dependencies to avoid these unnecessary computations. It requires $O(n)$ time in many cases but requires $O\left(n^2\right)$ time in the worst case.

Experimental results performed on the MatriCS platform showed that this sequential algorithm is $\times 18.93$ faster than the sequential algorithm of Yao (1982) and $\times 5.07$ faster than the CGM-based parallel solution of Myoupo and Kengne (2014a) on thirty-two processors. Therefore, it questions the core idea of partitioning techniques proposed by Kechid and Myoupo (2008b) and Myoupo and Kengne (2014a), which consists in scanning the $2n$ subproblems of the dependency graph to estimate the overall computational load to distribute tasks equitably onto processors (since two input data of the same size but different values have two different forms of the dependency graph).

Further work

Further research can be conducted to extend this work:

Parallelizing our fast sequential algorithm for the MCOP and the TCP problem
It would be interesting to undertake the challenge to propose a CGM-based parallel solution based on our sequential algorithm. Because of its speedup, the best partitioning technique of the dependency graph proposed in (Myoupo and Kengne, 2014a), which runs in $O(n \log p)$ time, must be completely revised. An idea can be to parallelize it. Another idea can be to skip the step of estimating the overall computational load before distributing the nodes equitably onto the processors. Distribute them randomly could be the best partitioning strategy.

Finding the ideal number of fragmentations before starting or during computations
It would be interesting to determine the ideal number of fragmentations before starting or during computations since this number depends on the architecture where the solution is executed. One idea will be to refer to machine learning tech-

niques to find it before starting according to the characteristics of parallel computer architectures. Another idea would be to create a dynamic irregular partitioning that subdivides the dynamic graph into blocks during the resolution of the problem according to the parallel computer architecture. It would also be interesting if these strategies took into account the optimization of other resources such as storage and energy consumption.

Exploiting the dynamic graph model of the OBST problem
It would be interesting to use our dynamic graph model of the OBST problem to develop a CGM-based parallel solution running in $O\left(n^2/p^2\right)$ execution time with $O(1)$ communication round. One idea can be to inspire ourselves from the work of Bradford (1994) and Higa and Stefanes (2012) to determine the critical nodes in the dynamic graph. From these nodes, it will be possible to avoid traversing certain paths that can never lead to the optimal solution.

Applying our partitioning techniques on other dynamic-programming problems
It would be interesting to propose CGM-based parallel solutions using our partitioning techniques to solve other non-serial polyadic dynamic-programming problems such as the context-free grammar parsing problem (Kasami, 1965; Younger, 1967) and the Nussinov RNA folding problem (Nussinov and Jacobson, 1980). To our knowledge, there is no CGM-based parallel solutions that solve these problems. Another serial monadic, serial polyadic, or non-serial monadic dynamic-programming problems can be also addressed such as the string editing problem (Alves et al., 2002; Lacmou et al., 2020) and the longest common subsequence problem (Garcia et al., 2003). The irregular partitioning technique has been applied to solve two variants of the longest common subsequence problem in (Kengne et al., 2022, 2020). It can be interesting to apply the four-splitting technique to improve this solution.

Applying our partitioning techniques on other parallel computer architectures
It would be interesting to apply our partitioning techniques on shared-memory architectures and GPU architectures. One idea can be to inspire the irregular partitioning technique and the loop tiling transformation to create a new transformation called *irregular loop tiling* that should mitigate the workload imbalance of threads. Indeed, the loop tiling transformation is used in many parallel solutions; for example to solve the Nussinov RNA folding problem in (Palkowski and Bielecki, 2017). Another idea can be to hybridize our solutions on distributed-memory architectures and those of Mabrouk (2016) on shared-memory architectures.

Bibliography

Alves, C. E. R., Cáceres, E. N., and Dehne, F. (2002). Parallel Dynamic Programming for Solving the String Editing Problem on A CGM/BSP. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures* (pp. 275–281). Winnipeg, Manitoba, Canada. https://doi.org/10.1145/564870.564916

Bellman, R. (1957). *Dynamic Programming* (1st ed.). Princeton University Press.

Bhattacharyya, S. S., and Murthy, P. K. (1995). Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams. In *Proceedings of the IEEE Workshop on VLSI Signal Processing* (pp. 177–186). Sakai, Kansai, Japan. https://doi.org/10.1109/vlsisp.1995.527489

Bielecki, W., Blaszynski, P., and Poliwoda, M. (2021). 3D Parallel Tiled Code Implementing a Modified Knuth's Optimal Binary Search Tree Algorithm. *Journal of Computational Science*, *48*(1). https://doi.org/10.1016/j.jocs.2020.101246

Biswas, G., and Mukherjee, N. (2021). Memory Optimized Dynamic Matrix Chain Multiplication Using Shared Memory in GPU. In *the 2021 International Conference on Distributed Computing and Internet Technology* (pp. 160–172). Bhubaneswar, Odisha, India. https://doi.org/10.1007/978-3-030-65621-8_10

Bradford, P. G. (1994). *Parallel Dynamic Programming* (Ph.D. Thesis). Indiana University.

Bradford, P. G., Rawlins, G. J., and Shannon, G. E. (1998). Efficient Matrix Chain Ordering In Polylog Time. *SIAM Journal on Computing*, *27*(2), 466–490. https://doi.org/10.1137/s0097539794270698

Cáceres, E. N., Mongelli, H., Loureiro, L., Nishibe, C., and Song, S. W. (2010). Performance Results of Running Parallel Applications on the Integrade. *Concurrency and Computation: Practice and Experience*, *22*(3), 375–393. https://doi.org/10.1002/cpe.1524

Chan, A., Dehne, F., Bose, P., and Latzel, M. (2008). Coarse Grained Parallel Algorithms for Graph Matching. *Parallel Computing*, *34*(1), 47–62. https://doi.org/10.1016/j.parco.2007.11.004

Collette, Y., and Siarry, P. (2002). *Optimisation Multiobjectif* (1st ed.). Eyrolles.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Craus, M. (2002). Parallel and Distributed Solutions for the Optimal Binary Search Tree Problem. In *the 2002 International Conference on Coding and Cryptology* (pp. 103–117). Mangalia, Dobroudja, Romania. https://doi.org/10.1007/3-540-47840-x_10

Culler, D., Karp, R., Patterson, D., Sahay, A., Erik, S. K., Santos, E., ... von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. *ACM SIGPLAN Notices*, *28*(7), 1–12. https://doi.org/10.1145/173284.155333

Czarnul, P. (2018). *Parallel Programming for Modern High Performance Computing Systems* (1st ed.). CRC Press.

Czech, Z. J. (2017). *Introduction to Parallel Computing* (1st ed.). Cambridge University Press.

Czumaj, A. (1993). Parallel Algorithm for the Matrix Chain Product and the Optimal Triangulation Problems. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science* (pp. 294–305). New York, USA. https://doi.org/10.1007/3-540-56503-5_30

Dehne, F. (2006). Guest Editor's Introduction. *Algorithmica*, *45*(3), 263–267. https://doi.org/10.1007/s00453-006-1213-2

Dehne, F., Fabri, A., and Rau-Chaplin, A. (1993). Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *Proceedings of the Ninth Annual Symposium on Computational Geometry* (pp. 298–307). San Diego, California, USA. https://doi.org/10.1145/160985.161154

Dehne, F., Ferreira, A., Cáceres, E., Song, S. W., and Roncato, A. (2002). Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica*, *33*(2), 183–200. https://doi.org/10.1007/s00453-001-0109-4

Deo, N. (1974). *Graph Theory with Applications to Engineering and Computer Science* (1st ed.). Prentice-Hall, Inc.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion With Graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/bf01386390

Diwan, T., and Tembhurne, J. (2019). A Parallelization of Non-Serial Polyadic Dynamic Programming On GPU. *Journal of Computing and Information Technology*, *27*(2), 55–66. https://doi.org/10.20532/cit.2019.1004579

El-Qawasmeh, E. (2004). Word Prediction Using a Clustered Optimal Binary Search Tree. *Information Processing Letters*, *92*(5), 257–265. https://doi.org/10.1016/j.ipl.2004.08.006

Feldmann, A., Gasser, O., Lichtblau, F., Pujol, E., Poese, I., Dietzel, C., ... Smaragdakis, G. (2020). The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *ACM Internet Measurement Conference* (pp. 1–18). New York, USA. https://doi.org/10.1145/3419394.3423658

Ferreira, A. (2001). Parallel Computing: Models. In *Encyclopedia of Optimization* (Vol. 547, pp. 1934–1939). Springer Boston. https://doi.org/10.1007/0-306-48332-7_380

Ferreira, A., and Morvan, M. (1997). Models for Parallel Algorithm Design: An Introduction. In *Parallel Computing in Optimization* (Vol. 7, pp. 1–26). Springer Boston. https://doi.org/10.1007/978-1-4613-3400-2_1

Floyd, R. W. (1962). Algorithm 97: Shortest Path. *Communications of the ACM*, *5*(6), 345. https://doi.org/10.1145/367766.368168

Flynn, M. J. (1966). Very High-Speed Computing Systems. *Proceedings of the IEEE*, *54*(12), 1901–1909. https://doi.org/10.1109/proc.1966.5273

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* (1st ed.). Addison-Wesley.

Fotso, L. P., Kengne, T. V., and Myoupo, J. F. (2010). Load Balancing Schemes for Parallel Processing of Dynamic Programming on BSP/CGM Model. In *the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 710–716). Las Vegas, Nevada, USA. https://hal.archives-ouvertes.fr/hal-01007650

Garcia, T., Myoupo, J. F., and Seme, D. (2003). A Coarse-Grained Multicomputer Algorithm for the Longest Common Subsequence Problem. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing* (pp. 349–356). Genova, Liguria, Italy. https://doi.org/10.1109/empdp

.2003.1183610

Gauss, C. F. (1866). Theoria Interpolationis Methodo Nova Tractata. In *Carl Friedrich Gauss Werke* (pp. 265–327). Göttingen, Basse-Saxe, Allemagne.

Glover, F. (1989). Tabu Search - Part I. *ORSA Journal on Computing*, *1*(3), 190–206. https://doi.org/10.1287/ijoc.1.3.190

Glover, F. (1990). Tabu Search - Part II. *ORSA Journal on Computing*, *2*(1), 4–32. https://doi.org/10.1287/ijoc.2.1.4

Godbole, S. S. (1973). On Efficient Computation of Matrix Chain Products. *IEEE Transactions on Computers*, *100*(9), 864–866. https://doi.org/10.1109/tc.1973.5009182

Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., and Tsantilas, T. (1999). Portable and Efficient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, *48*(7), 670–689. https://doi.org/10.1109/12.780876

Grama, A., Gupta, A., Karypis, G., and Kumar., V. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison Wesley.

Guibas, L. J., Kung, H. T., and Thompson, C. D. (1979). Direct VLSI Implementation of Combinatorial Algorithms. In *Proceedings of the Caltech Conference On Very Large Scale Integration* (pp. 509–525). Pasadena, California, USA. https://core.ac.uk/reader/9412580

Hager, G., and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers* (1st ed.). CRC Press.

Heideman, M. T., Johnson, D. H., and Burrus, C. S. (1984). Gauss and the History of the Fast Fourier Transform. *IEEE ASSP Magazine*, *1*(4), 14–21. https://doi.org/10.1109/massp.1984.1162257

Higa, D. R., and Stefanes, M. A. (2012). A Coarse-Grained Parallel Algorithm for the Matrix Chain Order Problem. In *Proceedings of the 2012 Symposium on High Performance Computing* (pp. 1–8). Orlando, Florida, USA. https://doi.org/10.5555/2338816.2338817

Hill, J. M., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., ... Bisseling, R. H. (1998). BSPlib: The BSP Programming Library. *Parallel Computing*, *24*(14), 1947–1980. https://doi.org/10.1016/s0167-8191(98)00093-3

Hu, T. C., and Shing, M. T. (1982). Computation of Matrix Chain Products. Part I.

SIAM Journal on Computing, *11*(2), 362–373. https://doi.org/10.1137/0211028

Hu, T. C., and Shing, M. T. (1984). Computation of Matrix Chain Products. Part II. *SIAM Journal on Computing*, *13*(2), 228–251. https://doi.org/10.1137/0213017

Huang, S. H. S., Liu, H., and Viswanathan, V. (1994). Parallel Dynamic Programming. *IEEE Transactions on Parallel and Distributed Systems*, *5*(3), 326–328. https://doi.org/10.1109/71.277784

Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, *40*(9), 1098–1101. https://doi.org/10.1109/jrproc.1952.273898

Hwang, K. (2008). *Advanced Computer Architecture: Parallelism, Scalability, Programmability* (2nd ed.). McGraw-Hill Higher Education.

Ibarra, O. H., Pong, T. C., and Sohn, S. M. (1991). Parallel Recognition and Parsing on the Hypercube. *IEEE Transactions on Computers*, *40*(6), 764–770. https://doi.org/10.1109/12.90253

Ito, Y., and Nakano, K. (2013). A GPU Implementation of Dynamic Programming For the Optimal Polygon Triangulation. *IEICE Transactions on Information and Systems*, *E96-D*(12), 2596–2603. https://doi.org/10.1587/transinf.e96.d.2596

JáJá, J. (1992). *An Introduction to Parallel Algorithms* (1st ed.). Addison Wesley.

Karpinski, M., Larmore, L., and Rytter, W. (1996). Sequential and Parallel Subquadratic Work Algorithms for Constructing Approximately Optimal Binary Search Trees. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 36–41). Philadelphia, Pennsylvania, USA. https://doi.org/10.5555/313852.313880

Karpinski, M., and Rytter, W. (1994). On a Sublinear Time Parallel Construction of Optimal Binary Search Trees. In *the 1994 International Symposium on Mathematical Foundations of Computer Science* (pp. 453–461). Košice, Slovakia. https://doi.org/10.1007/3-540-58338-6_92

Karypis, G., and Kumar, V. (1993). Efficient Parallel Mappings of a Dynamic Programming Algorithm: A Summary of Results. In *Proceedings of the Seventh International Parallel Processing Symposium* (pp. 563–568). Newport, California, USA. https://doi.org/10.1109/ipps.1993.262817

Kasami, T. (1965). *An Efficient Recognition and Syntax Analysis Algorithm*

for Context-Free Languages (Tech. Rep. No. AFCRL-65-758). Bedford, Massachusetts, USA: Air Force Cambridge Research Laboratory. https://www.ideals.illinois.edu/bitstream/handle/2142/74304/B4-257.pdf

Katagiri, T. (2019a). Basics of MPI Programming. In *The Art of High Performance Computing for Computational Science* (Vol. 1, pp. 27–44). Springer Singapore. https://doi.org/10.1007/978-981-13-6194-4_2

Katagiri, T. (2019b). Basics of OpenMP Programming. In *The Art of High Performance Computing for Computational Science* (Vol. 1, pp. 45–59). Springer Singapore. https://doi.org/10.1007/978-981-13-6194-4_3

Kechid, M., and Myoupo, J. F. (2008a). A Coarse Grain Multicomputer Algorithm Solving the Optimal Binary Search Tree Problem. In *the Fifth International Conference on Information Technology: New Generations* (pp. 1186–1189). Las Vegas, Nevada, USA. https://doi.org/10.1109/itng.2008.158

Kechid, M., and Myoupo, J. F. (2008b). An Efficient BSP/CGM Algorithm for the Matrix Chain Ordering Problem. In *the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 327–332). Las Vegas, Nevada, USA. https://hal.archives-ouvertes.fr/hal-03267731

Kechid, M., and Myoupo, J. F. (2009). A Coarse-Grain Multicomputer Algorithm for the Minimum Cost Parenthesization Problem. In *the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 480–486). Las Vegas, Nevada, USA. https://hal.archives-ouvertes.fr/hal-01007646

Kengne, T. V. (2014). *Solutions Parallèles Efficaces Sur Le Modèle CGM D'Une Classe de Problèmes Issus de la Programmation Dynamique* (Ph.D. Thesis). University of Picardie Jules Verne.

Kengne, T. V., Bogning, T. H., Akong, O. M., Myoupo, J. F., and Lacmou, Z. J. (2022). A Coarse-Grained Multicomputer Parallel Algorithm for the Sequential Substring Constrained Longest Common Subsequence Problem. *Parallel Computing*, *111*(1), 102927. https://doi.org/10.1016/j.parco.2022.102927

Kengne, T. V., and Lacmou, Z. J. (2019). An Efficient CGM-Based Parallel Algorithm for Solving the Optimal Binary Search Tree Problem Through One-to-All Shortest Paths in a Dynamic Graph. *Data Science and Engineering*, *4*(2), 141–156. https://doi.org/10.1007/s41019-019-0093-9

Kengne, T. V., and Myoupo, J. F. (2012). An Efficient Coarse-Grain Multicomputer Algorithm for the Minimum Cost Parenthesizing Problem. *The Journal of Supercomputing*, *61*(3), 463–480. https://doi.org/10.1007/s11227-011-0601-9

Kengne, T. V., Myoupo, J. F., and Dequen, G. (2016). High Performance CGM-based Parallel Algorithms for the Optimal Binary Search Tree Problem. *International Journal Grid High Performance Computing*, *8*(4), 55–77. https://doi.org/10.4018/ijghpc.2016100104

Kengne, T. V., Nkonjoh, N. A., Lacmou, Z. J., and Myoupo, J. F. (2020). Efficient CGM-Based Parallel Algorithms for the Longest Common Subsequence Problem With Multiple Substring-Exclusion Constraints. *Parallel Computing*, *91*(1), 102598. https://doi.org/10.1016/j.parco.2019.102598

Kessler, C., and Keller, J. (2007). Models for Parallel Computing: Review and Perspectives. *PARS-Mitteilunge*, *24*(1), 13–29. http://www.ida.liu.se/~chrke55/papers/modelsurvey.pdf

Kielmann, T., and Gorlatch, S. (2011). Bandwidth-Latency Models (BSP, LogP). In *Encyclopedia of Parallel Computing* (Vol. 796, pp. 107–112). Springer Boston. https://doi.org/10.1007/978-0-387-09766-4_189

Knuth, D. E. (1971). Optimum Binary Search Trees. *Acta Informatica*, *1*(1), 14–25. https://doi.org/10.1007/bf00264289

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

Kung, H. T. (1982). Why Systolic Architectures? *Computer*, *15*(1), 37–46. https://doi.org/10.1109/mc.1982.1653825

Kung, H. T., and Leiserson, C. E. (1978). Algorithms for VLSI Processor Arrays. In *Proceedings of a Symposium on Sparse Matrices and Their Applications* (pp. 256–282). Knoxville, Tennesse, USA. https://www.eecs.harvard.edu/~htk/publication/1980-introduction-to-vlsi-systems-kung-leiserson.pdf

Lacmou, Z. J., and Kengne, T. V. (2018). Speeding up CGM-Based Parallel Algorithm for Minimum Cost Parenthesizing Problem. In *the 2018 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 401–407). Las Vegas, Nevada, USA. https://hal.archives-ouvertes.fr/

hal-01900171

Lacmou, Z. J., Kengne, T. V., and Myoupo, J. F. (2021). A Fast Sequential Algorithm for the Matrix Chain Ordering Problem. *Concurrency and Computation: Practice and Experience*, *33*(24), e6445. https://doi.org/10.1002/cpe.6445

Lacmou, Z. J., Kengne, T. V., and Myoupo, J. F. (2022a). Coarse-Grained Multicomputer Parallel Algorithm Using the Four-Splitting Technique for the Minimum Cost Parenthesizing Problem. In *African Conference on Computer Science and Applied Mathematics* (pp. 1–12). Yaounde, Cameroon. https://hal.inria.fr/CARI2022/hal-03712194

Lacmou, Z. J., Kengne, T. V., and Myoupo, J. F. (2022b). Four-Splitting Based Coarse-Grained Multicomputer Parallel Algorithm for the Optimal Binary Search Tree Problem. *International Journal of Parallel, Emergent and Distributed Systems*, *37*(6), 659–679. https://doi.org/10.1080/17445760.2022.2102168

Lacmou, Z. J., Kengne, T. V., and Myoupo, J. F. (2022c). High-Performance CGM-Based Parallel Algorithms for Minimum Cost Parenthesizing Problem. *The Journal of Supercomputing*, *78*(4), 5306–5332. https://doi.org/10.1007/s11227-021-04069-9

Lacmou, Z. J., Tessa, M. G. C., and Kamga, Y. F. I. (2020). A CGM-Based Parallel Algorithm Using the Four-Russians Speedup for the 1-D Sequence Alignment Problem. In *African Conference on Computer Science and Applied Mathematics* (pp. 1–11). Thies, Senegal. https://hal.inria.fr/CARI2020/hal-02925791

Land, A. H., and Doig, A. G. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, *28*(3), 497–520. https://doi.org/10.2307/1910129

Lassous, I. G., Gustedt, J., and Morvan, M. (2000). Handling Graphs According to a Coarse Grained Approach: Experiments with PVM and MPI. In *Proceedings of the Seventh European PVM/MPI Users' Group Meeting* (pp. 72–79). Balatonfüred, Veszprém, Hungary. https://doi.org/10.1007/3-540-45255-9_13

Lee, H., Kim, J., Hong, S. J., and Lee, S. (2003). Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, *14*(4), 394–407. https://doi.org/10.1109/tpds.2003.1195411

Leighton, F. T. (1992). *Introduction to Parallel Algorithms and Architectures:*

Arrays, Trees, Hypercubes (1st ed.). Morgan Kaufmann.

Levesque, J., and Wagenbreth, G. (2010). *High Performance Computing : Programming and Applications* (1st ed.). CRC Press.

Lew, A., and Mauch, H. (2007). *Dynamic Programming: A Computational Tool* (1st ed.). Springer Verlag.

Li, G., Zhao, Q., Song, M., Du, D., Yuan, J., Chen, X., and Liang, H. (2019). Predicting Global Computing Power of Blockchain Using Cryptocurrency Prices. In *the 2019 International Conference on Machine Learning and Cybernetics* (pp. 1–6). Kobe, Kansai, Japan. https://doi.org/10.1109/icmlc48188.2019.8949188

Lin, S. S. (1994). A Chained-Matrices Approach For Parallel Computation of Continued Fractions and Its Applications. *Journal of Scientific Computing*, *9*(1), 65–80. https://doi.org/10.1007/bf01573178

Loshin, D. (2013). *Business Intelligence: The Savvy Managers Guide* (2nd ed.). Morgan Kaufmann.

Lovász, L. (2007). *Combinatorial Problems and Exercises* (2nd ed.). AMS Chelsea Publishing.

Mabrouk, B. B. (2016). *Application de la Programmation Dynamique Parallèle Pour la Résolution de Problèmes D'Optimisation Combinatoire* (Ph.D. Thesis). Université de Tunis El Manar.

Marvins, M. (1978). *Introduction to Modern Algebra* (1st ed.). Marcel Dekker.

McColl, W. F. (1995). Scalable Computing. In *Computer Science Today : Recent Trends and Developments* (Vol. 1000, pp. 46–61). Springer Verlag. https://doi.org/10.1007/bfb0015236

Midkiff, S. P. (2012). *Automatic Parallelization: An Overview of Fundamental Compiler Techniques* (1st ed.). Morgan Kaufmann.

Myoupo, J. F. (1992). Synthesizing Linear Systolic Arrays for Dynamic Programming Problems. *Parallel Processing Letters*, *2*(1), 97–110. https://doi.org/10.1142/s0129626492000222

Myoupo, J. F. (1993). Mapping Dynamic Programming Onto Modular Linear Systolic Arrays. *Distributed Computing*, *6*(3), 165–179. https://doi.org/10.1007/bf02242705

Myoupo, J. F., and Kengne, T. V. (2014a). An Efficient CGM-Based Parallel Algorithm Solving the Matrix Chain Ordering Problem. *International Journal of Grid and High Performance Computing*, *6*(2), 74–100. https://doi.org/10.4018/ijghpc.2014040105

Myoupo, J. F., and Kengne, T. V. (2014b). Parallel Dynamic Programming for Solving the Optimal Search Binary Tree Problem on CGM. *International Journal of High Performance Computing and Networking*, *7*(4), 269–280. https://doi.org/10.1504/ijhpcn.2014.062729

Nagaraj, S. (1997). Optimal Binary Search Trees. *Theoretical Computer Science*, *188*(1-2), 1–44. https://doi.org/10.1016/s0304-3975(96)00320-9

Nielsen, F. (2016). *Introduction to HPC with MPI for Data Science* (1st ed.). Springer Switzerland.

Nishida, K., Ito, Y., and Nakano, K. (2011). Accelerating the Dynamic Programming For the Matrix Chain Product on the GPU. In *Proceedings of the Second International Conference on Networking and Computing* (pp. 320–326). Osaka, Kansai, Japan. https://doi.org/10.1109/icnc.2011.62

Nussinov, R., and Jacobson, A. B. (1980). Fast Algorithm for Predicting the Secondary Structure of Single-Stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, *77*(11), 6309–6313. https://doi.org/10.1073/pnas.77.11.6309

Olariu, S. (2008). Transistional Issues: Fine-Grain Multicomputers. In *Handbook of Parallel Computing: Models, Algorithms and Applications* (Vol. 1, pp. 235–256). CRC Press. https://doi.org/10.1201/9781420011296-16

Ozdamli, F., and Ozdal, H. (2015). Life-long Learning Competence Perceptions of the Teachers and Abilities in Using Information-Communication Technologies. *Procedia - Social and Behavioral Sciences*, *182*(1), 718–725. https://doi.org/10.1016/j.sbspro.2015.04.819

Pacheco, P., and Malensek, M. (2021). *An Introduction to Parallel Programming* (2nd ed.). Morgan Kaufmann.

Palkowski, M., and Bielecki, W. (2017). Parallel Tiled Nussinov RNA Folding Loop Nest Generated Using Both Dependence Graph Transitive Closure and Loop Skewing. *BMC Bioinformatics*, *18*(1), 290. https://doi.org/10.1186/s12859-017-1707-8

Prim, R. C. (1957). Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal*, *36*(6), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP* (1st ed.). McGraw-Hill.

Raina, S. (1992). *Virtual Shared Memory: A Survey of Techniques and Systems* (Tech. Rep. No. CSTR-92-36). Bristol, United Kingdom, England: Department of Computer Science, University of Bristol. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.5133&rep=rep1&type=pdf

Ramanan, P. (1996). An Efficient Parallel Algorithm for the Matrix-Chain-Product Problem. *SIAM Journal on Computing*, *25*(4), 874–893. https://doi.org/10.1137/0225039

Rytter, W. (1988). On Efficient Parallel Computations for Some Dynamic Programming Problems. *Theoretical Computer Science*, *59*(3), 297–307. https://doi.org/10.1016/0304-3975(88)90147-8

Schmidt, B., González-Domínguez, J., Hundt, C., and Schlarb, M. (2017). *Parallel programming: Concepts and practice* (1st ed.). Morgan Kaufmann.

Schreier, O., and Sperner, E. (2011). *Introduction to Modern Algebra and Matrix Theory* (2nd ed.). Dover Publications.

Shyamala, K., Kiran, K. R., and Rajeshwari, D. (2017). Design and Implementation of GPU-Based Matrix Chain Multiplication Using C++AMP. In *Proceedings of the 2017 Second IEEE International Conference on Electrical, Computer and Communication Technologies* (pp. 1–6). Coimbatore, Tamil Nadu, India: IEEE. https://doi.org/10.1109/icecct.2017.8117870

Sitaram, D., and Manjunath, G. (2012). *Moving To The Cloud : Developing Apps in the New World of Cloud Computing* (1st ed.). Syngress.

Stanley, R. P. (2015). *Catalan Numbers* (1st ed.). Cambridge University Press.

Steffen, M., Divincenzo, D. P., Chow, J. M., Theis, T. N., and Ketchen, M. B. (2011). Quantum computing: An IBM perspective. *IBM Journal of Research and Development*, *55*(5), 1–13. https://doi.org/10.1147/jrd.2011.2165678

Sterling, T., Anderson, M., and Brodowicz, M. (2017). *High Performance Computing: Modern Systems and Practices* (1st ed.). Morgan Kaufmann.

Tan, G., Sun, N., and Gao, G. R. (2007). A Parallel Dynamic Programming Algorithm on a Multi-Core Architecture. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures* (pp. 135–144). San Diego, California, USA. https://doi.org/10.1145/1248377.1248399

Tang, D., and Gupta, G. (1995). An Efficient Parallel Dynamic Programming Algorithm. *Computers Mathematics with Applications*, *30*(8), 65–74. https://doi.org/10.1016/0898-1221(95)00138-o

Valiant, L. G. (1990). A Bridging Model for Parallel Computation. *Communications of the ACM*, *33*(8), 103–111. https://doi.org/10.1145/79173.79181

Valiant, L. G. (2011). A Bridging Model for Multi-Core Computing. *Journal of Computer and System Sciences*, *77*(1), 154–166. https://doi.org/10.1016/j.jcss.2010.06.012

Valiant, L. G., Skyum, S., Berkowitz, S., and Rackoff, C. (1983). Fast Parallel Computation of Polynomials Using Few Processors. *SIAM Journal on Computing*, *12*(4), 641–644. https://doi.org/10.1137/0212043

Vitorović, A., Tomašević, M. V., and Milutinović, V. M. (2014). Manual Parallelization Versus State-of-the-Art Parallelization Techniques: The SPEC CPU2006 as a Case Study. *Advances in Computers*, *92*(1), 203–251. https://doi.org/10.1016/b978-0-12-420232-0.00005-2

von Neumann, J. (1945). *First Draft of a Report on the EDVAC* (Tech. Rep. No. W-670-ORD-4926). Philadelphia, Pennsylvania, USA: Moore School of Electrical Engineering, University of Pennsylvania. http://web.mit.edu/sts.035/www/pdfs/edvac.pdf

Wagner, R. A., and Fischer, M. J. (1974). The String-to-String Correction Problem. *Journal of the ACM*, *21*(1), 168–173. https://doi.org/10.1145/321796.321811

Wah, B. W., and Li, G. J. (1988). Systolic Processing for Dynamic Programming Problems. *Circuits, Systems, and Signal Processing*, *7*(2), 119–149. https://doi.org/10.1007/bf01602094

Wani, M. A., and Ahmad, M. (2019). Statically Optimal Binary Search Tree Computation Using Non-Serial Polyadic Dynamic Programming on GPU's. *International Journal of Grid and High Performance Computing*, *11*(1), 49–70. https://doi.org/10.4018/ijghpc.2019010104

Wu, C.-l., and Feng, T. (1984). *Tutorial: Interconnection Networks for Parallel*

and Distributed Processing (1st ed.). IEEE Press.

Yao, F. F. (1982). Speed-up in Dynamic Programming. *SIAM Journal on Algebraic Discrete Methods*, *3*(4), 532–540. https://doi.org/10.1137/0603055

Yau, S.-T., and Lu, Y. Y. (1993). Reducing the Symmetric Matrix Eigenvalue Problem to Matrix Multiplications. *SIAM Journal on Scientific Computing*, *14*(1), 121–136. https://doi.org/10.1137/0914008

Younger, D. H. (1967). Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, *10*(2), 189–208. https://doi.org/10.1016/s0019-9958(67)80007-x

Yzelman, A. J. N., and Roose, D. (2014). High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, *25*(1), 116–125. https://doi.org/10.1109/tpds.2013.31

Zhong, H.-S., Wang, H., Deng, Y.-H., Chen, M.-C., Peng, L.-C., Luo, Y.-H., … Pan, J.-W. (2020). Quantum Computational Advantage Using Photons. *Science*, *370*(6523), 1460–1463. https://doi.org/10.1126/science.abe8770

APPENDIX A

List of Publications

International journals

Kengne Tchendji Vianney and Lacmou Zeutouo Jerry (2019). An Efficient CGM-Based Parallel Algorithm for Solving the Optimal Binary Search Tree Problem through One-to-all Shortest Paths in a Dynamic Graph. *Data Science and Engineering*, *4*(2), 141-156. https://doi.org/10.1007/s41019-019-0093-9

Lacmou Zeutouo Jerry, Kengne Tchendji Vianney, and Myoupo Jean Frédéric (2021). A Fast Sequential Algorithm for the Matrix Chain Ordering Problem. *Concurrency and Computation: Practice and Experience*, *33*(24), e6445. https://doi.org/10.1002/cpe.6445

Lacmou Zeutouo Jerry, Kengne Tchendji Vianney, and Myoupo Jean Frédéric (2022). High-Performance CGM-Based Parallel Algorithms for Minimum Cost Parenthesizing Problem. *The Journal of Supercomputing*, *78*(4), 5306-5332. https://doi.org/10.1007/s11227-021-04069-9

Lacmou Zeutouo Jerry, Kengne Tchendji Vianney, and Myoupo Jean Frédéric (2022). Four-Splitting Based Coarse-Grained Multicomputer Parallel Algorithm for the Optimal Binary Search Tree Problem. *International Journal of Parallel, Emergent and Distributed Systems*, *37*(6), 659-679. https://doi.org/10.1080/17445760.2022.2102168

International conference

Lacmou Zeutouo Jerry and Kengne Tchendji Vianney (2018). Speeding up CGM-Based Parallel Algorithm for Minimum Cost Parenthesizing Problem. In *the 2018 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 401-407). Las Vegas, Nevada, USA. https://hal.archives-ouvertes.fr/hal-01900171

Lacmou Zeutouo Jerry, Kengne Tchendji Vianney, and Myoupo Jean Frédéric (2022). Coarse-Grained Multicomputer Parallel Algorithm Using the Four-Splitting Technique for the Minimum Cost Parenthesizing Problem. In *African Conference on Computer Science and Applied Mathematics* (pp. 1-12), Yaounde, Cameroon. https://hal.inria.fr/CARI2022/hal-03712194