ARIMA

épisciences

Coarse-grained multicomputer parallel algorithm using the four-splitting technique for the minimum cost parenthesizing problem

Jerry LACMOU ZEUTOUO[*1], **Vianney KENGNE TCHENDJI**[1], **Jean-Frédéric MYOUPO**[2]

[1]Department of Mathematics and Computer Science, University of Dschang, Cameroon
[2]Computer Science Lab-MIS, University of Picardie Jules Verne, France

*E-mail : jerrylacmou@gmail.com

DOI : 10.46298/arima.11217
Submitted on 19 avril 2023 - Published on 27 octobre 2023
Volume : **38** - Year : **2023**
Special Issue : **CARI 2022**
Editors : *Mathieu Roche, Clémentin Tayou Djamegni, Nabil Gmati, Amel Ben Abda, Marcellin Nkenlifack*

Abstract

Dynamic programming is a technique widely used to solve several combinatory optimization problems. A well-known example is the minimum cost parenthesizing problem (MPP), which is usually used to represent a class of non-serial polyadic dynamic-programming problems. These problems are characterized by a strong dependency between subproblems. This paper outlines a coarse-grained multicomputer parallel solution using the four-splitting technique to solve the MPP. It is a partitioning technique consisting of subdividing the dependency graph into subgraphs (or blocks) of variable size and splitting large-size blocks into four subblocks to avoid communication overhead caused by a similar partitioning technique in the literature. Our solution consists in evaluating a block by computing and communicating each subblock of this block to reduce the latency time of processors which accounts for most of the global communication time. It requires $\mathcal{O}\left(n^3/p\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ 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.

Keywords

coarse-grained multicomputer ; dynamic programming ; dynamic graph ; irregular partitioning ; four-splitting

I INTRODUCTION

Given a chain of symbols (characters, numbers, matrices, or objects), the minimum cost parenthesizing problem (MPP) consists in finding the parenthesizing that will minimize the cost of the computations involved on this chain. This problem appears in the literature under several variants depending on the kind of entities in the chain to parenthesize and the treatment to perform. The most popular variant is the matrix chain ordering problem consisting in finding the parenthesizing that minimizes the cost of the product of a chain of matrices [4]. Moreover, the MPP is

typically used in the literature to represent a class of non-serial polyadic dynamic-programming problems that can be modeled by Equation (1). Godbole [1] proposed the so-called *generic sequential algorithm* to solve this problem in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space.

Some research have investigated the parallelization of this sequential algorithm on different parallel computing models. On realistic models of parallel machines, Nishida et al. [6] presented an efficient parallel implementation of the generic sequential algorithm on GPU architectures. Ito and Nakano [8] accelerated this solution by partitioning the generic sequential algorithm 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. Shyamala et al. [11] accelerated the computation time through C++ high-performance accelerated massive parallel code. More recently, Diwan and Tembhurne [13] 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 [14] 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, Mabrouk [10] designed solutions based on loop transformations. She showed that the associative expression evaluation technique with the loop interchange transformation provides efficient parallel solutions. On distributed-memory architectures, many researchers proposed their parallel solutions on the coarse-grained multicomputer (CGM) model.

The CGM model introduced in [2] is the most suitable model to design parallel solutions that are not too dependent on a specific architecture like the systolic and hypercube models. It enables to formalize the performance of a parallel algorithm only with the input data size and the number of processors. A CGM-based parallel algorithm consists in successively repeating a computation round and a communication round until the problem is solved. In each computation round, processors perform local computations on their data using the best sequential algorithm. They exchange data through the network in each communication round. All information sent from one processor to another is wrapped into a single long message to minimize the overall message overhead. For designing a CGM-based parallel solution, the standard methodology in the literature consists of subdividing the dependency graph into subgraphs (or blocks) of same size, then fairly distributing these blocks among processors, and finally computing them in a suitable evaluation order. Designers' efforts tend to reduce the number of communication rounds and the overall computation time to produce an efficient parallel solution [2].

Kengne et al. [9] showed that there is a trade-off between the minimization of the number of communication rounds and the load-balancing of processors when the dependency graph is partitioned into blocks of the same size :

1. when it is subdivided into small-size blocks like in [5], the load difference between processors is small if one processor has one more block than another (however, the number of communication round will be high);
2. when it is subdivided into large-size blocks like in [7], the number of communication rounds of the corresponding algorithm is reduced since there are few blocks (however, the load of processors will be unbalanced).

Lacmou and Kengne [12] tackled this trade-off by proposing the irregular partitioning technique. It consists in subdividing the dependency graph into blocks of variable size. It ensures that the blocks of the first steps (or diagonals) are 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 enable processors to stay active longer. These blocks are fairly distributed over processors to minimize their idle time and balance the load between them. It requires $\mathcal{O}(n^3/p)$ total execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds, where 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. Experimental results showed that the irregular partitioning technique significantly reduced the total execution time compared regular partitioning techniques proposed in [5, 7, 9].

Nevertheless, this technique also induces an important latency time of processors which accounts for most of the global communication time. In fact, it does not enable processors to start evaluating small-size blocks as soon as the data they need are available; yet they are usually available before the end of the evaluation of large-size blocks. To solve this issue, Lacmou et al. [15] proposed the k-block splitting technique to reduce this latency time. This technique consists in splitting the large-size blocks into a set of smaller-size blocks called *k-blocks*. Thus, a single processor evaluates a block by computing and communicating each k-block contained in this block to enable processors to start the evaluation of k-blocks as soon as possible. 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 the CGM-based parallel solution using this technique as the communication overhead raises the latency time of processors.

In this paper, we propose the four-splitting technique to reduce the latency time of processors caused by the irregular partitioning technique. This technique consists in splitting the large-size blocks into four small-size blocks (or subblocks). Hence, evaluating a block by a single processor will consist of computing and communicating each subblock contained in this block. The goal is the same as the k-block splitting technique, that is, enables processors to start the evaluation of blocks as soon as possible. Our CGM-based parallel solution requires $\mathcal{O}(n^3/p)$ total execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds. Experimental results showed a good agreement with theoretical predictions. Our CGM-based parallel solution using this technique were better than those using the irregular partitioning technique and the k-block splitting technique. These results also showed that when the number of fragmentations is small, it is preferable to use k-block splitting technique than the four-splitting technique when the data size and the number of processors become very high.

The remainder of this paper is structured as follows. Section 2 summarizes background knowledge of the MPP (the dynamic-programming formulation, the Godbole sequential algorithm, and the dynamic graph model of Bradford [3]). Then, Section 3 presents our CGM-based parallel solution using the four-splitting technique. Section 4 outlines the experimental results archived, and finally Section 5 concludes this work.

II BACKGROUND

2.1 Dynamic-programming formulation

The dynamic-programming formulation of the MPP is defined by :

$$Cost[i,j] = \begin{cases} Init(i) & \text{if } 1 \leq i = j \leq n, \\ \min\limits_{i \leq k < j} \{Cost[i,k] + Cost[k+1,j] + F(i,k,j)\} & \text{if } 1 \leq i < j \leq n. \end{cases} \quad (1)$$

Algorithm 1 Godbole's sequential algorithm

```
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;
```

In Equation (1), n is the problem size. $Cost[i, j]$ corresponds to the value of the optimal solution of the subproblem (i, j). This value is obtained 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 $Init(i)$.

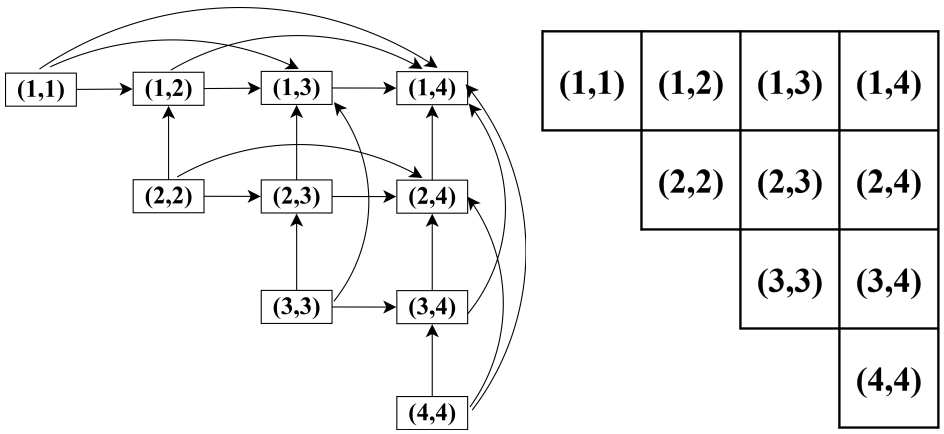
2.2 Godbole's sequential algorithm

A solution based on the exhaustive search of all possible combinations will be poor because, for a problem of size n, the number of combinations is exponential in n [4]. To solve the MPP, Godbole [1] proposed the first polynomial-time sequential algorithm running in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space. It became the standard algorithm for solving all problems that can be formulated by Equation (1) because the structure and the complexity of this algorithm are independent of the functions F and $Init$. Algorithm 1 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 dependency graph (or task graph), as shown in Figure 1(a) 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 1 uses a bottom-up approach to compute the optimal solutions of subproblems. The values of these solutions are stored in the DP table depicted in Figure 1(b).

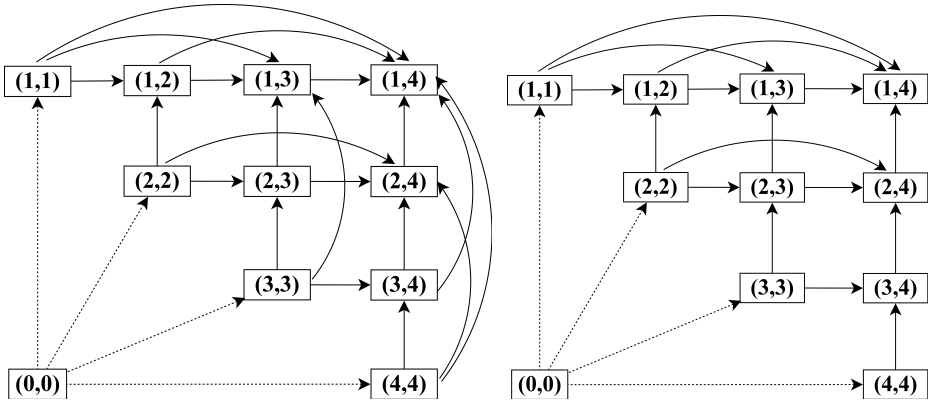
2.3 Dynamic graph model

Bradford [3] showed that the MPP can be solved through the shortest path algorithms on weighted dependency graph by designing a graph model called *dynamic graph* for the problem that can be formulated by Equation (1). For a problem of size n, this graph is denoted by D_n. 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 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). Indeed, each path from the root to an edge node corresponds to one of the possible parenthesizes in a dynamic graph. Thus, the shortest path corresponds to the optimal parenthesizing. Figure



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

Figure 1: Task graph and the dynamic-programming table used to compute $Cost[1,4]$



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

Figure 2: Dynamic graphs D_4 and D'_4 for a problem of size $n = 4$

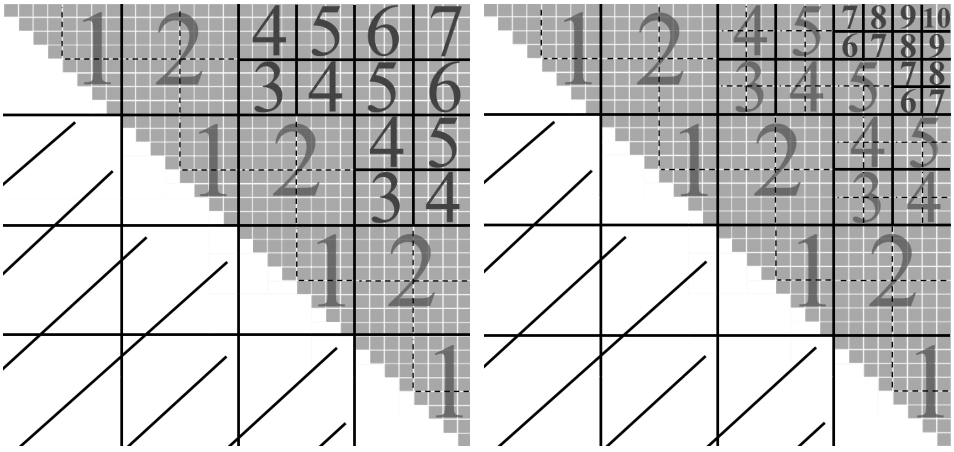
2(a) shows a dynamic graph D_n for $n = 4$. It has the same dependency graph form representing the dependency between subproblems depicted in Figure 1(a), with an additional node $(0, 0)$.

Given a dynamic graph D_n, Bradford [3] proved that 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 property is fundamental because it helps 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 solution is a subgraph of D_n denoted by D'_n, in which the set of edges from (i, k) to (i, j) and from $(k + 1, j)$ to (i, j) is removed. Figure 2(b) shows the dynamic graph D'_4.

III OUR CGM-BASED PARALLEL SOLUTION

3.1 Dynamic graph partitioning

The irregular partitioning technique consists in subdividing the shortest path matrix into submatrices (blocks) of varying size to enable 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



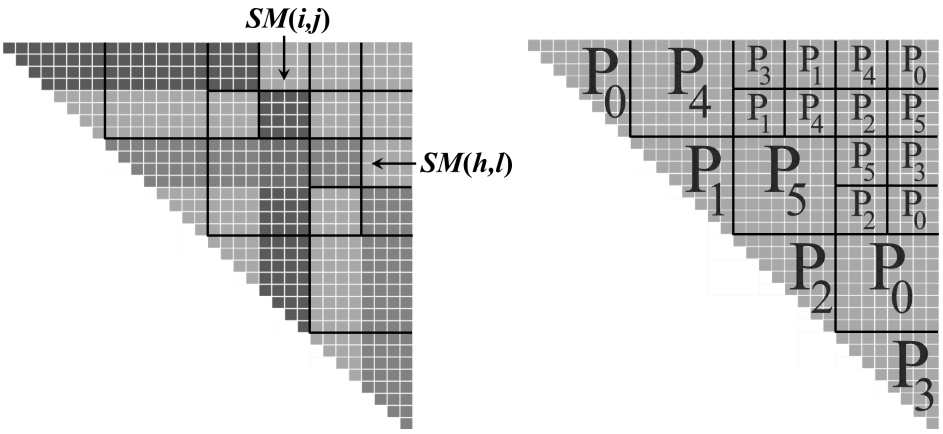
((a)) $p \in \{5, 6, 7, 8\}$ and $k = 1$ ((b)) $p \in \{5, 6, 7, 8\}$ and $k = 2$

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

half of the first one through the block fragmentation technique. This technique aims to reduce the block size by dividing it into four subblocks. 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 diagonal 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 enable 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 lth level of fragmentation have been subdivided l times, $0 \leq l \leq k$.

The four-splitting technique consists in splitting the large-size blocks into four smaller-size blocks (or subblocks) after performing k fragmentations to reduce the latency time of processors. 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 to enable processors to start the evaluation of blocks as soon as possible. Hence, evaluating a block by a processor will consist of computing and communicating each subblock contained in this block.

By denoting $f(p) = \lceil\sqrt{2p}\rceil$, $\theta(n, p) = \lceil n/f(p)\rceil$, and $\theta(n, p, l) = \lceil\theta(n, p)/2^l\rceil$, 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 3(a) and 3(b) depict two scenarios of this partitioning for $n = 32$, $k \in \{1, 2\}$, and $p \in \{5, 6, 7, 8\}$. The number in each block represents the diagonal in which it belongs.



((a)) Blocks' dependency ((b)) Snake-like mapping

Figure 4: Dependencies of two blocks and snake-like mapping when $p = 6$ and $k = 1$

3.2 Blocks' dependency and mapping onto processors

Figure 4(a) illustrates an example of dependencies of two blocks $SM(i, j)$ and $SM(h, l)$. The most shaded blocks are required to evaluate them. It can be noticed that the evaluation of shortest paths for blocks of the same diagonal can be carried out in parallel. Indeed, the dependency relationship between blocks proved that those on the same diagonal are independent [12, 15].

We use a snake-like mapping scheme [7] to enable 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. We reiterate this process until all processors are used, starting with processor 0 and traveling through the blocks with a "snake-like" path. Figure 4(b) depicts a mapping scheme on six processors. This mapping enables processors to remain active as soon as possible. It also ensures the load balancing because it equally distributes small and large size blocks among processors. However, it does not optimize communications.

3.3 Our CGM-based parallel algorithm

Our CGM-based parallel algorithm based on the four-splitting technique to solve the MPP is given by Algorithm 2. This 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 [5, 7]. Thus, evaluating the shortest path cost of nodes of a block belonging to the diagonal d starts at the diagonal $\lceil d/2 \rceil$. After computing each subblock containing in a block, it is immediately communicated to processors that need these subblocks for updating or for finalizing the computations of values in next steps.

Theorem 1:
Our CGM-based parallel solution based on the four-splitting technique to solve the MPP runs in $\mathcal{O}(n^3/p)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds in the worst case.

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

- three subblocks at the first diagonal of blocks;
- four subblocks from the diagonal of blocks 2 to $\lfloor S/2 \rfloor - 1$;

Algorithm 2 Our CGM-based parallel algorithm based on the four-splitting technique

1: $d \leftarrow f(p) + k \times (\lceil f(p)/2 \rceil + 1)$;
2: **for** $u = 1$ to d **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 $(u+1, \ldots, \min\{2 \times (u-1), d\})$ **do**
7: **Update** the shortest path costs to nodes of the subblock ρ';

- 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 $\mathcal{O}\left(\frac{n^3}{2^{3(l+1)} \times (2p)^{3/2}}\right) = \mathcal{O}\left(\frac{n^3}{8^{l+1} \times p\sqrt{p}}\right)$ local computation time. So, the evaluation of each diagonal of blocks required :

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

The number of communication rounds is equal to :

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

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

IV EXPERIMENTAL RESULTS

4.1 Experimental setups

We compare our new CGM-based parallel solution denoted by *4s* with our previous best solution [15] denoted by *kbyk*. They have been implemented in C[1] and executed on the operating system CentOS Linux release 7.6.1810. We used the cluster dolphin of the MatriCS platform of the University of Picardie Jules Verne[2]. Each compute node is made of two Intel Xeon Processor E5-2680 V4 (35M Cache, 2.40 GHz), and each of them consists of 14 cores. This cluster

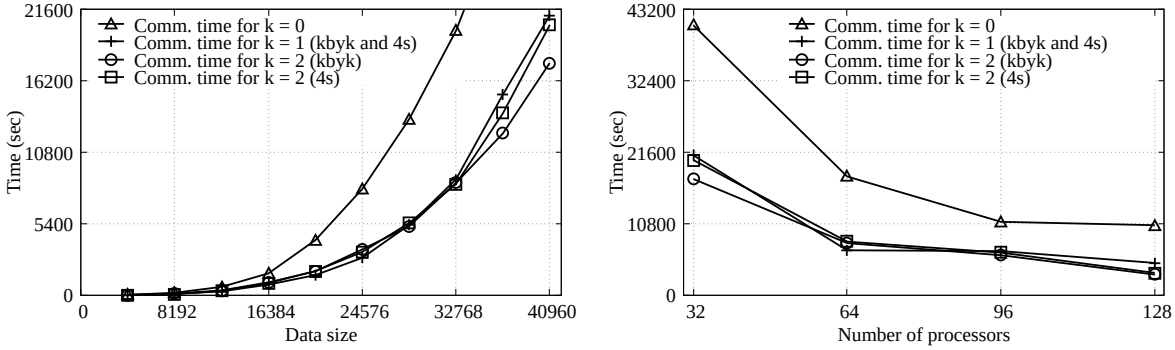
[1]The source codes are available : https://github.com/compiii/CGM-Sol-for-MPP.
[2]https://www.matrics.u-picardie.fr

consists of 60 compute nodes interconnected with OmniPath links providing 100Gbps throughput, and divided into 48 thin nodes with 48 $\times$ 128GB of RAM and 12 thick nodes with 12 $\times$ 512GB of RAM. These algorithms have been executed on five thin nodes. The MPI library (OpenMPI version 1.10.4) has been used for inter-processor communication. We present the results following the different values of the triplet (n, p, k). n is the data size, with values in the set $\{4096, \ldots, 40960\}$. p is the number of processors, with values in the set {32, 64, 96, 128}. k is the number of fragmentations of blocks performed, with values in the set {0, 1, 2, 3, 4}. When $k = 0$, our solution is reduced to the one in [7]. *kbyk* and *4s* are similar when $k = 1$.

4.2 Evolution of the global communication time

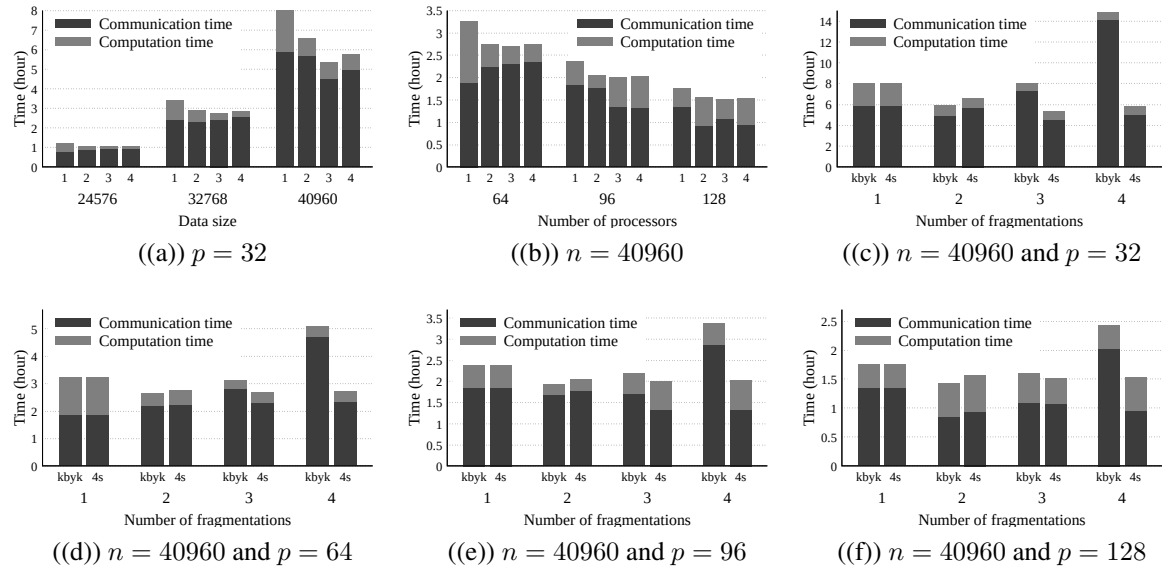
Figures 5(a) and 5(b) compare the global communication time of *kbyk* and *4s* while solving the MPP by performing one and two fragmentations. Figure 5(a) shows that from $n = 4096$ to 24576, the global communication time of *kbyk* and *4s* is better when performing one fragmentation than when performing 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 6(a), 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 5(a) 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 5(b), 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 enable 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 6(a) 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 6(d), 6(e), and 6(f) 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 6(a) and 6(b) show that in general the global communication time and the overall computation time of *4s* gradually decrease while solving the MPP by performing four fragmentations successively. Nevertheless, Figures 6(a) and 6(b) show that the global communication time of *4s* degrades from the fourth fragmentation because on thirty-two processors when $n = 40960$,



((a)) $p = 32$ ((b)) $n = 40960$

Figure 5: Global communication time for $n \in \{4096, \ldots, 40960\}$, $p \in \{32, \ldots, 128\}$, and $k \in \{0, 1, 2\}$



((a)) $p = 32$ ((b)) $n = 40960$ ((c)) $n = 40960$ and $p = 32$

((d)) $n = 40960$ and $p = 64$ ((e)) $n = 40960$ and $p = 96$ ((f)) $n = 40960$ and $p = 128$

Figure 6: Comparison of the overall computation time and the global communication time for $n \in \{24576, 32768, 40960\}$, $p \in \{32, \ldots, 128\}$, and $k \in \{1, \ldots, 4\}$

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 fragmentation, the small subblocks which are close to the optimal solution (and thus require a high evaluation time using Godbole's sequential algorithm) 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.

4.3 Evolution of the total execution time

Figures 6(a), 6(b), 6(c), 6(d), 6(e), and 6(f) show the total execution time of *kbyk* and *4s* while solving the MPP by performing one, two, three, and four fragmentations. The global communication time has a huge impact on the total execution time. 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;

- from $n = 28800$ on thirty-two processors, the total execution time of *kbyk* is better than that of *4s* when performing two fragmentations;
- 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;
- 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;
- 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.

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 abruptly degrade when the number of fragmentations increases.

V CONCLUSION AND FUTURE RESEARCH DIRECTIONS

In this paper, we presented a CGM-based parallel solution for solving the minimum cost parenthesizing problem using the four-splitting technique. This technique avoids 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. It requires $\mathcal{O}\left(n^3/p\right)$ total execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds, where n is the input data size and p is the number of processors. The experimental results archived showed a good agreement with our theoretical predictions. It would be interesting to propose CGM-based parallel solutions using this four-splitting technique to solve other non-serial polyadic dynamic-programming problems such as the context-free grammar parsing problem and the Nussinov RNA folding problem. To our knowledge, there is no CGM-based parallel solutions that solve these problems. It would be also interesting to apply our partitioning technique on shared-memory architectures and GPU architectures.

REFERENCES

[1] S. S. Godbole. "On Efficient Computation of Matrix Chain Products". In: *IEEE Transactions on Computers* 100.9 (1973), pages 864–866.

[2] F. Dehne, A. Fabri, and A. Rau-Chaplin. "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers". In: *Proceedings of the Ninth Annual Symposium on Computational Geometry*. San Diego, California, USA, 1993, pages 298–307.

[3] P. G. Bradford. "Parallel Dynamic Programming". Ph.D. Thesis. Indiana University, 1994.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.

[5] M. Kechid and J. F. Myoupo. "A Coarse-Grain Multicomputer Algorithm for the Minimum Cost Parenthesization Problem". In: *the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications*. PDPTA'09. Las Vegas, Nevada, USA, 2009, pages 480–486.

[6] K. Nishida, Y. Ito, and K. Nakano. "Accelerating the Dynamic Programming For the Matrix Chain Product on the GPU". In: *Proceedings of the Second International Conference on Networking and Computing*. Osaka, Kansai, Japan, 2011, pages 320–326.

[7] T. V. Kengne and J. F. Myoupo. “An Efficient Coarse-Grain Multicomputer Algorithm for the Minimum Cost Parenthesizing Problem”. In: *The Journal of Supercomputing* 61.3 (2012), pages 463–480.

[8] Y. Ito and K. Nakano. “A GPU Implementation of Dynamic Programming For the Optimal Polygon Triangulation”. In: *IEICE Transactions on Information and Systems* E96-D.12 (2013), pages 2596–2603.

[9] T. V. Kengne, J. F. Myoupo, and G. Dequen. “High Performance CGM-based Parallel Algorithms for the Optimal Binary Search Tree Problem”. In: *International Journal Grid High Performance Computing* 8.4 (2016), pages 55–77.

[10] B. B. Mabrouk. “Application de la Programmation Dynamique Parallèle Pour la Résolution de Problèmes D’Optimisation Combinatoire”. Ph.D. Thesis. Université de Tunis El Manar, 2016.

[11] K. Shyamala, K. R. Kiran, and D. Rajeshwari. “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*. Coimbatore, Tamil Nadu, India: IEEE, 2017, pages 1–6.

[12] Z. J. Lacmou and T. V. Kengne. “Speeding up CGM-Based Parallel Algorithm for Minimum Cost Parenthesizing Problem”. In: *the 2018 International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, Nevada, USA, 2018, pages 401–407.

[13] T. Diwan and J. Tembhurne. “A Parallelization of Non-Serial Polyadic Dynamic Programming On GPU”. In: *Journal of Computing and Information Technology* 27.2 (2019), pages 55–66.

[14] G. Biswas and N. Mukherjee. “Memory Optimized Dynamic Matrix Chain Multiplication Using Shared Memory in GPU”. In: *the 2021 International Conference on Distributed Computing and Internet Technology*. Bhubaneswar, Odisha, India, 2021, pages 160–172.

[15] Z. J. Lacmou, T. V. Kengne, and J. F. Myoupo. “High-Performance CGM-Based Parallel Algorithms for Minimum Cost Parenthesizing Problem”. In: *The Journal of Supercomputing* 78.4 (2022), pages 5306–5332.