Speeding up CGM-Based Parallel Algorithm for Minimum Cost Parenthesizing Problem

Jerry Lacmou Zeutouo, Vianney Kengne Tchendji

Speeding up CGM-Based Parallel Algorithm for Minimum Cost Parenthesizing Problem

Jerry Lacmou Zeutouo and Vianney Kengne Tchendji
Department of Mathematics and Computer Science, University of Dschang, Dschang, Cameroon

Abstract— *Parallelization of dynamic programming algorithms is a very well studied topic. This paper presents a parallel algorithm on the Coarse Grained Multicomputer model (CGM for short) for solving the Optimal String Parenthesizing Problem or Minimum Cost Parenthesizing Problem. In the previous best CGM algorithm for this problem, which uses regular partitioning of the dynamic graph, it is difficult to simultaneously optimize load-balancing of the processors and minimization of the communication time in the same algorithm. In this paper, we try to bring a solution to these two contradictory objectives. We propose a new technique of irregular partitioning of this graph which allows processors to stay active as long as possible. This promotes load-balancing and thus minimizes the overall computation time of processors. This also reduces their latency, which is the largest part of the overall communication time.*

Keywords: Dynamic Programming, Parallel Algorithm, Coarse Grained Multicomputer, Dynamic Graph, Irregular Partitioning

1. Introduction

Dynamic programming (DP for short) is a technique which can be applied to solve several combinatorial optimization problems. The idea in DP is to order the computations of solutions of sub-problems in such a way that each of them is compute once. The study of dependency between sub-problems shows that they are organizable on a multi levels Directed Acyclic Graph (DAG for short). DP problems are often solved through shortest path problems on weighted DAGs. In this paper we are interested in a typical DP problems class that can be modelled by Equation (1). It is the class of all combinatorial optimization problems that uses a graphic model proposed by Bradford [1], called *dynamic graph*, to solve the Minimum Cost Parenthesizing Problem.

The Minimum Cost Parenthesizing Problem (MPP for short) consists of finding, given a chain of characters, the parenthesizing that will minimize the cost of the computations involved. It is also called Optimal String Parenthesizing problem. The classical sequential algorithm for this problem requires $\mathcal{O}(n^3)$ time steps and $\mathcal{O}(n^2)$ memory space [2].

The parallelisation of the classical sequential algorithm was extensively treated by the community of parallel processing researchers for the different computing parallel models. Ibarra et al. [3] presented a solution in $\mathcal{O}(n^2)$ time steps on $\mathcal{O}(n)$ processors on a hypercube. Guibas et al. [4] proposed a bidirectional systolic algorithm on $\mathcal{O}(n^2)$ processors with $\mathcal{O}(n)$ running time. Karypis and Kumar [5] mapped a systolic table onto a mesh-connected array of size n^2. Myoupo [6] proposed a technique for mapping the MPP to a linear systolic array. Huang et al. [7] improved the PRAM solution in 1994, reducing the number of processors to $\mathcal{O}(n^6/\log^5 n)$.

In this paper, we tackle the problem of parallelizing MPP algorithm on the BSP/CGM (*Bulk Synchronous Parallel model/Coarse Grained Multicomputer*) [8][9]. CGM seems best suited for the design of algorithms that are not too dependent on an individual architecture. A BSP/CGM machine is a set of p processors. Each having its own local memory of size m (with $\mathcal{O}(m) \gg \mathcal{O}(1)$) and connected to a router able to deliver messages in point-to-point fashion. A BSP/CGM algorithm consists in alternating local computations and global communication rounds. Each communication round consists in routing a single h-relation with $h = \mathcal{O}(m)$. A CGM computation/communication round corresponds to a BSP super-step with communication cost $g \times m$ [10]. g is the cost of the communication of a word in the BSP model. To produce an efficient BSP/CGM algorithm, designers effort tend to maximize speedup and minimize the number of communication rounds (ideally independent from the problem size, and, constant in the optimum).

Tchendji and Myoupo [11][12], have shown that for MPP, load balancing of the processors and minimization of the communication time are two contradictory objectives when the corresponding tasks graph is partitioned into sub-graphs (or blocks) of the same size. Indeed, as each block of the graph is fully evaluated by a single processor, we have the following two scenarios:

1) for minimizing communication rounds, the blocks must be of large size, thus the number of blocks of the graph is minimized, and therefore the number of communication rounds of the corresponding algorithm is reduced. It is the case of the CGM-based paralell algorithm proposed by Tchendji and Myoupo [12] which requires $\lceil\sqrt{2p}\rceil$ communication rounds and $\mathcal{O}(n^3/p)$ time steps on p processors;
2) for load-balancing of the processors, the blocks must be of smaller size. Thus, if a processor has one more block than another processor, because blocks are of small size, the load difference between the processors will also be low. It is the case of the CGM-based

paralell algorithm proposed by Kechid and Myoupo [13] which requires $\mathcal{O}(p)$ communication rounds and $\mathcal{O}(n^3/p)$ time steps on p processors.

Their solution gives to the final user the choice to optimize one criterion or another according to the parameters g (granularity of their model) and p (number of processors) [11].

The main drawback of this solution is the contradictory optimization criterion caused by the regular size of blocks of the tasks graph when partitioning. Indeed, the final user cannot optimize more than one criterion. The second drawback is idleness of processors. Indeed, the number of blocks of the diagonals become quickly lower than p; and so several processors cannot be active at the same time. From there, over time, there is more and more idle processors (one more after each step). Yet, the blocks of the upper levels are those which induce biggest loads. Therefore this idleness is at the origin of the unbalance load between the processors and even of their latency.

Our contribution is to propose a BSP/CGM parallel algorithm based on a dynamic graph for solving the MPP which try to bring a solution to the contradictory objectives of the minimization of the communication time and the load balancing of the processors in this type of graph. It use our new technique of irregular partitioning of this graph which allows processors to stay active as long as possible. This promotes load-balancing and thus minimizes the overall computation time of processors. This also reduces their latency, which is the largest part of the overall communication time.

The remainder of this paper is organized as follow: In Section 2, we give the definition of the MPP and a short review of its sequential solution. Section 3 present a dynamic graph model for MPP. Section 4 is devoted to the presentation of our BSP/CGM algorithm. The experimental results are analyzed in Section 5. The last section concludes our work.

2. Minimum cost parenthesizing problem

The Minimum Cost Parenthesizing Problem (MPP for short), also called Optimal String Parenthesizing problem, consists of finding the parenthesizing given a chain of characters that will minimize the cost of the computations involved. This problem represents the class of all combinatorial optimization problems that can be modelled by Equation (1), such that :

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

In Equation (1), n is the size of the problem, and the values $Init(i)$ and $F(i,k,j)$ are known or can be easily computed. The optimal solution of the problem at hand is given by $Cost(1,n)$. In the function F, $F(i,k,j)$ gives for the sub-problem (i,j), the cost of combining the two optimal solutions of (i,k) and $(k+1,j)$ in order to produce optimal solution for (i,j). This function, called *union function*, depends in the intrinsic characteristics of concerned problem.

The number of possible solutions for a problem of size n, is exponential in n: $\Omega\left(4^n/n^{3/2}\right)$. Thus, a solution based on the direct exhaustive search is very poor. In the early 1970s, several researchers noted the presence of characteristics, like *optimality* and the *overlapping sub-problems*, in the Matrix Chain Ordering Problem (MCOP for short). Applying the technique of dynamic programming, in 1973, Godbole [2] proposed the first polynomial time solution for the MCOP. It requires $\mathcal{O}(n^3)$ time steps and $\mathcal{O}(n^2)$ memory space. The structure and the complexity of this solution being independent of the MCOP union function F, it has become the standard algorithm for solving all problems that can be modelled by equation (1). That is why this algorithm is often called *generic algorithm* in the literature. Its general structure is given in Algorithm 1.

Algorithm 1: The general structure of Godbole algorithm.

```
1 for d = 1 to n do
2     for i = 1 to n − d + 1 do
3         Cost(i, i + d − 1) ← ∞;
4         for k = i to i + d − 2 do
5             temp ← Cost(i, k) + Cost(k + 1, i + d −
              1) + F(i, k, i + d − 1);
6             if temp < Cost(i, i + d − 1) then
7                 Cost(i, i + d − 1) ← temp;
```

The calculation of $Cost(1,n)$ involves the resolution of all sub-problems $(i,j)/1 \leq i \leq j \leq n$. The study of the dependencies between the sub-problems shows that they are organized on a n-level Directed Acyclic Graph (DAG for short) with the following properties:

- each node represents a sub-problem. A level d is a set of nodes representing the set of sub-problems $(i, i+d-1)/1 \leq i \leq n-d+1$;
- an arc from a node representing (i,j) to the one representing (i',j') indicates that computing of $Cost(i',j')$ depends on $Cost(i,j)$. The node with no outgoing arcs represents the original problem, $Cost(1,n)$.

Figure 1 shows the form of the n-level DAG (with $n=4$). In fact, algorithm 1 solves sub-problems corresponding to the multi-level DAG using a bottom-up approach, level by level from the lowest level. It is called a *dynamic programming algorithm* [2]. For each sub-problem it solves, the value of its optimal solution is save in a table (a (n,n) triangular matrix) called the *dynamic programming table*.

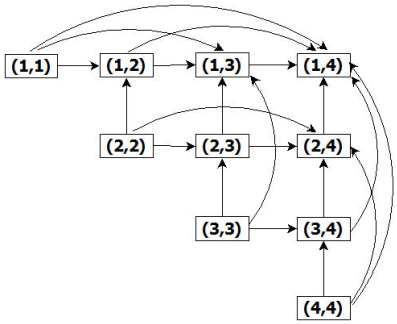


Fig. 1: 4-levels DAG schematizing dependencies between different sub-problems involved in the calculation of $Cost(1,4)$.

3. Dynamic graph model

The dynamic programming problems are often solved through the shortest paths problems on weighted DAGs. For the problems class described in Section 2, a graphic model, called *dynamic graph*, appeared in [1]. It is noted D_n for a problem of size n. Figure 2 shows the dynamic graph D_n for $n = 4$. It has the same form of DAG representing the dependency between sub-problems presented above, with an additional node $(0,0)$. Bradford [1] shown that the calculation of $Cost(i,j)$ is equivalent to search in D_n the shortest path from node $(0,0)$ to (i,j). In a dynamic graph, each path from the root to an edge node corresponds to one of the possible parenthesizings. Therefore, the shortest path corresponds to the optimal parenthesizing.

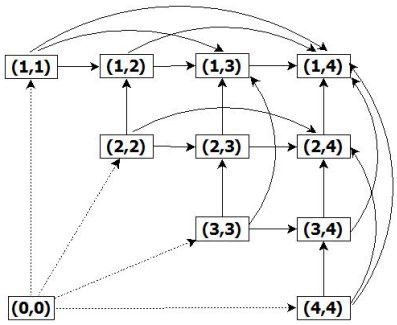


Fig. 2: Dynamic graph D_4.

Therefore, it is straightforward to prove that the Godbole algorithm presented above (Algorithm 1) is equivalent to calculating the shortest paths from $(0,0)$ to the other vertices in a dynamic graph D_n, incrementally, diagonal after diagonal, from left to right. A table called the *tasks graph* or *Shortest Paths Matrix*, $SP(n,n)$, is defined by setting $SP(i,j)$ to be the matrix of the shortest path from node $(0,0)$ to $(i,j)/1 \leq i \leq j \leq n$.

Given a problem of size n and its corresponding dynamic graph D_n, it was shown in [1] that:

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

This is a fundamental element of our BSP/CGM algorithm. It helps to avoid computation redundancy of the shortest path costs in D_n.

4. CGM algorithm for MPP

In of this section, we present our CGM algorithm for MPP which try to bring a solution to the contradictory objectives mentioned in Introduction. Firstly we partition the shortest path matrix (or the D_n graph) into sub-matrices (or sub-graphs) of varying size (irregular size) in Section 4.1. Next, we study the dependencies between sub-problems in Section 4.2 and distribute the blocks onto processors in Section 4.3. Finally, we present the CGM algorithm in Section 4.4.

4.1 Task graph partitioning

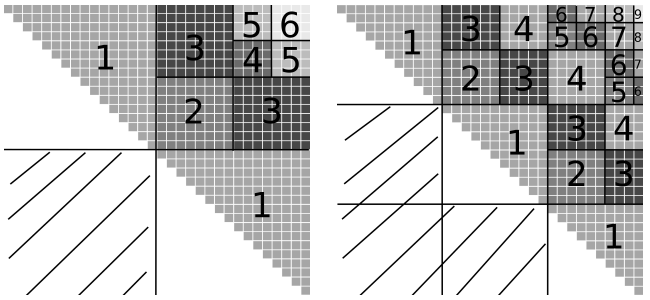
The idea is to start the subdivision on the largest diagonal of the shortest path matrix with blocks of large sizes, in order to minimize the number of communications. We reiterate this same partitioning on the following diagonals. Then, since the number of blocks per diagonal quickly becomes smaller than the number of processors; when a diagonal reaches half of the first diagonal of blocks, fragmentation is carried out (that is to say, the size of the blocks is reduced) to catch up (or exceed by one notch) the number of blocks of the first diagonal of blocks in order to increase the number of blocks of these diagonals and allow a maximum of processors to remain active. This minimizes the idleness of the processors, and thus promotes their loads balancing. After k fragmentations, we no longer modify the size of the blocks and the rest of the partitioning becomes traditional because an excessive fragmentation of the blocks would lead to a drastic increase in the number of communication rounds.

Formally, denote $f(p) = \lceil\sqrt{2p}\rceil$, $\theta(n,p) = \lfloor n/f(p)\rfloor$ and $\theta(n,p,k) = \lceil\theta(n,p)/2^k\rceil$, we partition the shortest path matrix $SP(n,n)$ into sub-matrices or blocks ($SM(i,j)$ for short). $SM(i,j)$ is a $\theta(n,p,k) \times \theta(n,p,k)$ matrix at the k^{th} fragmentation, i.e. at each fragmentation we divide the current size of the blocks into 4. Equation (2) shows the entries of the table $SP(n,n)$ delimiting a block $SM(i,j)/1 \leq i \leq j \leq n$.

$$C = \begin{pmatrix} SP(i,j-S+1) & \cdots & SP(i,j) \\ \vdots & \cdots & \vdots \\ SP(i+S-1,j-S+1) & \cdots & SP(i+S-1,j) \end{pmatrix}$$

where $C = SM(i,j)$ and $S = \theta(n,p,k)$ (2)

Figure 3 shows two scenarios of this partitioning for $n = 32$, $k = 2$ and $p \in \{2,3,4\}$. The number in each block represents the diagonal in which it belongs.



(a) For $n = 32$, $k = 2$ and $p = 2$, the shortest path matrix is partitioned into nine blocks.

(b) For $n = 32$, $k = 2$ and $p \in \{3, 4\}$, the shortest path matrix is partitioned into twenty four blocks.

Fig. 3: Partitioning for $n = 32$, $k = 2$ and $p \in \{2, 3, 4\}$.

Remark 1: .

1) the blocks of the first diagonal are upper triangular matrices of $\theta(n, p)$ lines and $\theta(n, p)$ columns;
2) a block is full if it is a non-triangular matrix of size $\theta(n, p, k) \times \theta(n, p, k)$;
3) in general, blocks in the last column of blocks are not full (this is shown in figure 3b);
4) one fragmentation increases $\lceil f(p)/2 \rceil + 1$ diagonals;
5) when $f(p)$ is odd, the number of blocks in a diagonal after each fragmentation exceeds by one notch the number of superblocks (the larger blocks) of the first diagonal (this is illustrated in figure 3b where there are 3 blocks in diagonal 1 and 4 blocks in the diagonal 3).

We can now derive the following lemmas 1 and 2.

Lemma 1: The number of blocks of the dynamic graph (the shortest path matrix) after partitioning is a function of k and is:

$$C = (k-1) \times \frac{(S+1)(S+2\beta) - \lceil \frac{S}{2} \rceil \left(\lceil \frac{S}{2} \rceil - 1 \right)}{2} + (S+1)(S+\beta) + \left\lceil \frac{S}{2} \right\rceil \frac{1 - \lceil \frac{S}{2} \rceil}{2}$$

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

Proof: After partitioning, there is exactly $S(S+1)/2 - \lceil S/2 \rceil \ (\lceil S/2 \rceil + 1)/2$ superblocks. Depending on the parity of S, we have the following two scenarios:

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 the diagonals from the first to the $(k-1)^{th}$ fragmentation (for example diagonals 2, 3 and 4 on figure 3b). This number increases by $S(S+1)/2 + \lceil S/2 \rceil$ blocks after the k^{th} fragmentation;
2) when S is odd, the principle is the same as when it is even, except that here fragmentation increases $(S+1)$ additional blocks on the initial block numbers (see point 5 of remark 1).

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

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

■

Lemma 2: Our strategy of irregular partitioning of the dynamic graph induces $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ diagonals of blocks when the blocks undergo k successives fragmentations.

Proof: If no fragmentation is performed when partitioning the task graph (i.e. if $k = 0$), then there are $f(p)$ diagonals. Suppose there are k fragmentations. One fragmentation increases $\lceil f(p)/2 \rceil + 1$ 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 the 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 equivalent to $\lceil f(p)/2 \rceil + 1$ diagonals (for example the diagonals 2, 3 and 4 on figure 3b). 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. ■

4.2 Blocks dependancy

Lemma 3 (Nodes dependency): To find the cost of the shortest path to a node (i, j) in graph D_n, it is necessary to know the cost of shortest path of each one of nodes $(i, i), \ldots, (i, j-1)$ and $(i+1, j), \ldots, (j, j)$.

Proof: See proof in [13]. ■

Lemma 4 (Weights of jumps to a block): To compute the weights of jumps from nodes of $SP(i, k)$ to nodes of $SP(i, j)/1 \leq i \leq k \leq j \leq n$, we only need the costs of the shortest paths to nodes of $SP(k - \theta(n, p) + 2, j)$.

Proof: See proof in [12]. ■

From Lemma 3 and Lemma 4, we have the following:

Theorem 2 (Blocks dependency): Let u an integer such that $u = \lceil (j-i)/\theta(n, p, k) \rceil$. The costs of the shortest paths to every node in blocks $SM(i, j - \theta(n, p, k))$, $SM(i, j - 2 \times \theta(n, p, k))$,..., $SM(i, j - (u-1) \times \theta(n, p, k))$, and $SM(i + \theta(n, p, k), j)$, $SM(i + 2 \times \theta(n, p, k), j)$,..., $SM(i + (u-1) \times \theta(n, p, k), j)$ are necessary for the evaluations of shortest paths to the nodes of block $SM(i, j)$ where is found at the k^{th} fragmentation.

Figure 4 shows the dependency of $SM(i, j)$ and $SM(h, l)$ on the other blocks (in bold). The evaluation of the shortest paths for nodes of different blocks of the same diagonal

can be carried out in parallel because the dependency relationship between blocks shows that those on the same diagonal are independent.

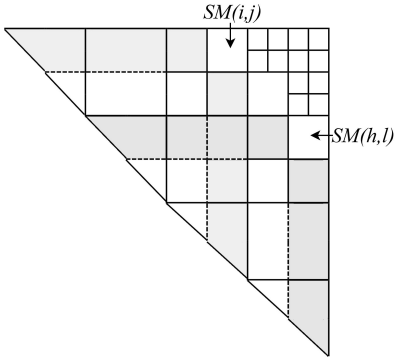


Fig. 4: Two blocks dependency representation.

Remark 2: In order to minimize the amount of data exchanged between the processors in the communication phases, each processor communicates only the sub-set of its block, corresponding to the size of the target block.

4.3 Mapping blocks onto processors

In this mapping, all blocks of the main diagonal are assigned from the leftmost upper corner to the rightmost lower corner. This process is renewed until all processors have been used, starting with processor 1 and traveling through the blocks with a "*snake like*" path, as shown in figure 5.

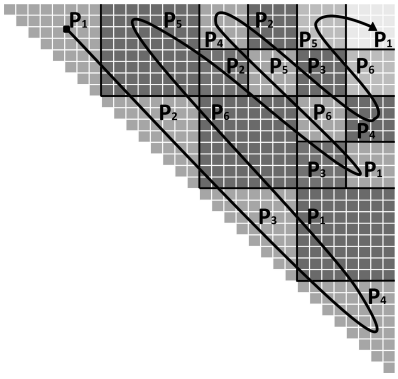


Fig. 5: Snake-like mapping of blocks onto processors for $p = 6$, $k = 1$ and $n = 32$.

This mapping allows to some processors to evaluate at most one block more than the others. Also, it contribute to load-balancing between processors because the blocks are evenly distributed among the processors. However, it has a drawback: due to the snake-like data distribution onto processors, communications are not minimized.

Lemma 5: With the snake-like distribution of the blocks onto processors, each processor has to evaluate at most $(3k+2)$ blocks. k is the number of fragmentations of blocks performed.

Proof: Depending on the parity of $f(p)$, we have the following two scenarios:

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

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

4.4 CGM algorithm

Owing to the dependencies of the data between the blocks of the graph D_n, the evaluation of these blocks must be done according to a well adapted order. Indeed, the values of the shortest paths to the vertices of the blocks of a diagonal d cannot be obtained before those of the vertices contained in each of the blocks on which they depend on the preceding diagonals (the diagonals $1, 2, \ldots, d-1$ according to Theorem 2). However, the nature of these calculations (minimum values) allows to start them before the end of the evaluation of the blocks of the diagonal $d-1$.

Theorem 3: After the computation of 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: See proof in [12]. ■

Our BSP/CGM algorithm is a succession of $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ similar steps (iterations). In each of them, the blocks of a diagonal (called *on line diagonal*) are evaluated in parallel. We start processing on the first diagonal, followed by the second and so on till the last. The overall structure of our algorithm given by the Algoritm 2, allows at the end of the process, to obtain the value of $Cost(1, n)$.

Our approach consists of progressively compute the values of shortest paths to each node of sub-graphs (blocks). The evaluation of the shortest path to a node of a block of diagonal d starts at diagonals $\lceil d/2 \rceil$, the numbering of diagonal goes from left to right and ranges from 1 to $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$. At the end of the computation of blocks on diagonal d (step 1 of Algoritm 2 at iteration d), each block is forwarded (step 3 of Algoritm 2 at iteration d) to processors that need these blocks for updating (step 2 of Algoritm 2) or for finalizing (step 1 of Algoritm 2) the computations of values in next iterations. In fact at iteration $d + 1$ two tasks have to be done :

1) for each input of a block of a diagonal m, $(d + 2 \leq m \leq 2d)$, some new values may be computed and an

Algorithm 2: The general structure of our BSP/CGM algorithm.

Data: Shortest paths matrix ($SP(n,n)$), initialized to 0 and stored on a CGM(n,p).
Result: For each processor the values of the shortest paths for each sub-problem it holds.

for $d = 1$ **to** $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ **do**
- Step 1 : **Finalization** of computation of the values of the shortest paths to nodes in blocks of diagonal d ;
- Step 2: **Update** the values of the shortest paths to each block of diagonals $(d+1, d+2, \ldots, \min\{2 \times (d-1), f(p) + k \times (\lceil f(p)/2 \rceil + 1)\})$;
- Step 3: **Communication** of block $SM(i,j)$ of on line diagonal to the processors that detain upper blocks and right blocks ;

update of its temporary values is done (path relaxation principle [14]);

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

These processes are repeated at each iteration until the iteration $f(p) + k \times (\lceil f(p)/2 \rceil + 1)$ where only step 1 of Algorithm is carried out. At iteration 1, only steps 1 and 3 are executed. It is not difficult to observe that computing the values of the shortest paths (in which jumps from blocks $SM(i,k)$ are involved) to block $SM(i,j)$ is equivalent to the sequential matrix-multiplication[1] $(+, \min)$ of the two matrices $SM(i,k)$ and $SM(k - \theta(n,p,k) + 2, j)$. Thus, the procedures for phases 1 and 2 of the algorithm are given below.

Algorithm 3: Finalization of computation of block $SM(i,j)$.

for $d = (j - i - \theta(n,p,k))$ **to** $(j-i)$ **do**
- **foreach** *node s of diagonal d belonging to* $SP(i,j)$ **do**
  - $SP(s) \leftarrow$ minimum ($SP(s)$, weight of paths whose final edge are jumps coming from block $SP(i, i + \theta(n,p,k))$, weights of paths whose final edges are internal jumps, weights of paths whose final edges are unit edges).

After the $(j-i)^{th}$ iteration of Algorithm 2, the only paths which remain to be valuated for nodes in $SP(i,j)$ are those whose last edge is (1) either a unit edge (vertical or horizontal), (2) either an horizontal jump which comes from an internal node in $SP(i,j)$ or (3) an horizontal jump which comes from a node in $SP(i,i)$. In any case, the computation of the weight induced by each of these paths (due to these edges) to a node (i', j') of block $SP(i,j)$ needs the value of shortest path from a node (e', f') of block $SP(i,j)$ such that $f' - e' < j' - i'$.

[1]Matrix-multiplication $(+, min)$ is a multiplication of matrices in which the operations of multiplication and summation are replaced by addition and the minimum, respectively.

Algorithm 4: Update of block $SM(i,j)$ at $(h+1)^{th}$ iteration, $\lceil (j - i/2) \rceil + 1 \leq h \leq j - i + 1$.

for $d = (j - i - \theta(n,p,k))$ **to** $(j-i)$ **do**
- $M_1 \leftarrow$ matrix-multiplication$(+, \min)$ $(SP(i, h+i-1),\ SP(h+i-1, j))$;
- $M_2 \leftarrow$ matrix-multiplication$(+, \min)$ $(SP(i, j-h+1),\ SP(j-h+1, j))$;
- $SP(i,j) \leftarrow \min\{SP(i,j), M_1, M_2\}$;

In cases (1) and (2), this value is necessary to compute the shortest path of the departure node of the last edge. In case (3), this value is necessary for computing the weight of the last edge. Therefore, the procedure for step 1 is a classical algorithm of the shortest path in block $SP(i,j)$, in which each node can receive a simple edge or a jump from an internal node in block $SP(i,j)$ or in block $SP(i,i)$.

Lemma 6: The time complexity of the Algorithms 3 and 4 is $\mathcal{O}(n^3/(2p)^{3/2})$.

Proof: The finalization in step 1 uses the Godbole sequential algorithm on problems of size $\theta(n,p)$, which is a size of super-blocks. The update in step 2 uses the sequential multiplication of $\theta(n,p) \times \theta(n,p)$ matrices. So, its time complexity in the worst case is in $\mathcal{O}(n^3/(2p)^{3/2})$. ■

Theorem 4: The CGM algorithm runs, in the worst case, in $\mathcal{O}\left(n^3/p\right)$ time steps per processor and $\lceil \sqrt{2p} \rceil + k \times \left(\left\lceil \lceil \sqrt{2p} \rceil / 2 \right\rceil + 1\right)$ communication rounds. k is the number of fragmentations of blocks performed.

Proof: At the k^{th} fragmentation, the Godbole sequential algorithm (Alogrithm 3) and the sequential multiplication of two matrices (Algorithm 4) used in the local computation phases of our CGM algorithm requires $\mathcal{O}\left(n^3/(2^{2k} \times (2p)^{2/2})\right) = \mathcal{O}\left(n^3/(4^k \times (2p))\right)$ local computations for the evaluation of each block of a diagonal of blocks. So, we have for each processor (see proof of lemma 5):

$$D = \mathcal{O}\left(\frac{n^3}{2p}\right) \times \left(1 + \frac{3}{4} + \frac{3}{4^2} + \cdots + \frac{3}{4^{k-1}} + \frac{4}{4^k}\right) = \mathcal{O}\left(\frac{n^3}{p}\right)$$

We conclude that this algorithm requires $\mathcal{O}\left(n^3/p\right)$ local computations time on each processor. The number of rounds of communication is derived from lemma 2. ■

Remark 3: When $k = 0$, our CGM algorithm reduces to the one in [12], with $\mathcal{O}\left(n^3/p\right)$ time steps per processor and $\lceil \sqrt{2p} \rceil$ communication rounds.

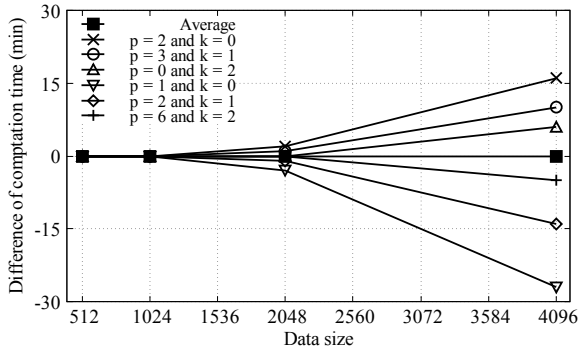
5. Experimental results

We implemented our algorithm on the cluster dolphin of the MATRICS platform of the University of Picardie Jules Verne using 60 computation nodes, and which each node is an Intel Xeon Processor E5-2680 V4 (35M Cache, 2.40GHz).

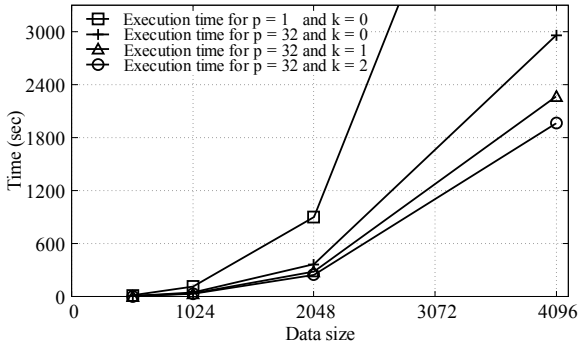
The C programming language is used, on the operating system CentOS Linux release 7.4.1708. The inter-processor communication is implemented with the MPI library (Open-MPI version). To explore the performance of our algorithm, the results presented here are derived from its execution for different values of the triplet (n, p, k), where:

- n is the problem size (number of data), with values in the set $\{512, 1024, 2048, 4096\}$;
- p is the number of processors, with values in the set $\{1, 2, 5, 8, 25, 28, 32\}$;
- k is the number of fragmentations of blocks performed, with values in the set $\{0, 1, 2\}$.

Figure 6a presents the curves of different loads compared to their average load (each value of k has its own load average). It show that irregular partitioning of the dynamic graph balances the load between the processors better than regular partitioning of this graph. It is due to the progressive reduction of size of the blocks which allows processors to stay active as long as possible. Thus, it minimize the overall computations times of processors. This also reduces their latency and then minimizes the communication time. Figure 6b shows that all this reduces the overall execution time of the algorithm as the number of fragmentations increases.



(a) Load difference relative to the average load for $p = 8$, $n \in \{512, \ldots, 4096\}$ and $k \in \{0, 1, 2\}$.



(b) Total execution time in function of n with $p \in \{1, 32\}$, $n \in \{512, \ldots, 4096\}$ and $k \in \{0, 1, 2\}$.

Fig. 6: Load difference relative to the average load and total execution time in function of n and p.

6. Conclusion and future works

In this paper, we have presented an efficient parallel algorithm on the BSP/CGM model for solving the MPP on p processors. It use our new technique of irregular partitioning of the dynamic graph to try to bring a solution to the contradictory objectives of the minimization of the communication time and the load balancing of the processors in this type of graph. It runs, in the worst case, in $\mathcal{O}\left(n^3/p\right)$ time steps per processor and $\lceil\sqrt{2p}\rceil + k \times \left(\left\lceil\lceil\sqrt{2p}\rceil/2\right\rceil + 1\right)$ communication rounds. The experimental results show a good agreement with theoretical predictions.

The irregular partitioning technique of the tasks graph may be applicable to other dynamic programming problems in the same class of the MPP as Matrix Chain Ordering Problem. This work is also left for future works.

References

[1] P. G. Bradford, "Parallel Dynamic Programming," Indiana Univ., 1994.

[2] S. S. Godbole, "On efficient computation of matrix chain products," *IEEE Trans. on Computers*, vol. 100, pp. 864–866, 1973.

[3] O. H. Ibarra, T. C. Pong, and S. M. Sohn, "Parallel recognition and parsing on the hypercube," *IEEE Trans. on Computers*, vol. 40, pp. 764–770, 1991.

[4] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. of the Caltech Conference On Very Large Scale Integration*, 1979, pp. 509–525.

[5] G. Karypis, and V. Kumar, "Efficient parallel mappings of a dynamic programming algorithm," in *Proc. of 7th inter. parallel processing symposium*, 1993, pp. 563–568.

[6] J. F. Myoupo, "Mapping dynamic programming onto modular linear systolic arrays," *Distributed Computing*, vol. 6, pp. 165–179, 1993.

[7] S. Huang, H. Liu, and V. Viswanathan, "Parallel dynamic programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pp. 326–328, 1994.

[8] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, pp. 103–111, 1990.

[9] F. Dehne, A. Fabri, and A. Rau-chaplin, "Scalable parallel computational geometry for coarse grained multicomputers," *Inter. Journal on Computational Geometry and Applications*, vol. 6, pp. 379–400, 1996.

[10] F. Dehne, A. Ferreira, E. Caceres, S. Song, and A. Roncato, "Efficient parallel graph algorithms for coarse grained multicomputers and BSP," *Algorithmica*, vol. 33, pp. 183–200, 2000.

[11] V. K. Tchendji, J. F. Myoupo, and G. Dequen, "High Performance CGM-based Parallel Algorithms for the Optimal Binary Search Tree Problem," *Int. J. Grid High Perform. Comput.*, vol. 8, pp. 55–77, 2016.

[12] V. K. Tchendji and J. F. Myoupo, "An efficient coarse-grain multicomputer algorithm for the minimum cost parenthesizing problem," *J. Supercomput.*, vol. 61, pp. 463–480, 2012.

[13] M. Kechid and J. F. Myoupo, "A Course-Grain Multicomputer Algorithm for the Minimum Cost Parenthesization Problem," in *Proc. of the Inter. Conf. on PDPTA 2009*, 2009, pp. 480–486.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, Third Edition," 3rd ed. The MIT Press, 2009.