An Efficient CGM-Based Parallel Algorithm for Solving the Optimal Binary Search Tree Problem Through One-to-All Shortest Paths in a Dynamic Graph

Vianney Kengne Tchendji¹ · Jerry Lacmou Zeutouo¹

Received: 15 May 2018 / Revised: 28 March 2019 / Accepted: 6 June 2019 / Published online: 17 June 2019
 © The Author(s) 2019

Abstract

The coarse-grained multicomputer parallel model (CGM for short) has been used for solving several classes of dynamic programming problems. In this paper, we propose a parallel algorithm on the CGM model, with p processors, for solving the optimal binary search tree problem (OBST problem), which is a *polyadic non-serial* dynamic programming problem. Firstly, we propose a dynamic graph model for solving the OBST problem and show that each instance of this problem corresponds to a one-to-all shortest path problem in this graph. Secondly, we propose a CGM parallel algorithm based on our dynamic graph to solve the OBST problem through one-to-all shortest paths in this graph. It uses our new technique of irregular partitioning of the dynamic graph to try to bring a solution to the well-known contradictory objectives of the minimization of the communication time and the load balancing of the processors in this type of graph. Our solution is based on Knuth's sequential solution and required $\mathcal{O}\left(\frac{n^2}{p}\right)$ time steps per processor and $\lceil \sqrt{2p} \rceil + k \times \left(\left\lceil \frac{\lceil \sqrt{2p} \rceil}{2} \right\rceil + 1 \right)$ communication rounds. Integer k is a parameter used in the partitioning technique of our algorithm. This new CGM algorithm performs better than the previously most efficient solution, which uses regular partitioning of the tasks graph.

Keywords Coarse-grained multicomputer · Optimal binary search tree · Dynamic graph model · Irregular partitioning

1 Introduction

Dynamic programming is a technique that can be applied to solve several combinatorial optimization problems. The idea in dynamic programming is to order the computations of solutions of subproblems in such a way that each of them is computed once. To do this, the problem is subdivided into steps (levels). Each step is composed of several subproblems and has a resolution strategy and a number of associated states. The solution of each of the subproblems of a given step depends only on those of the subproblems belonging to the preceding steps. Thus, the complexity of a dynamic programming problem depends on the intensity

of the dependence between the subproblems of the different steps. Relying on the nature of the dependence of subproblems, we distinguish two types of multilevel dynamic programmings [23]:

1. *serial* dynamic programming, where the solution of a subproblem of a given level depends exclusively on the solutions of the subproblems of the immediately preceding level and
2. *non-serial* dynamic programming, where the solution of a subproblem of a given level depends on the solutions of the subproblems of several preceding levels.

Moreover, a dynamic programming problem is said to be *monadic* if the corresponding cost function has only one term of recurrence (see Eq. 1), and it is said to be *polyadic* otherwise. The problems that use dynamic programming technique induce a lot of calculations. In this paper, we study the optimal binary search tree problem (OBST for short) which is a *polyadic non-serial* dynamic programming

✉ Vianney Kengne Tchendji
 vianneykengne@yahoo.fr

Jerry Lacmou Zeutouo
 jerrylacmou@gmail.com

¹ Department of Mathematics and Computer Science,
 University of Dschang, Dschang, Cameroon

problem in which there is a strong dependency and an irregularity of calculation load between the subproblems.

Given n keys $(K_1, K_2, \dots, K_n)$, the access probabilities of each key, and those occurring in the gap between two successive keys, an optimal binary search tree for this set of keys is one which has the smallest search cost. The OBST problem is to construct an optimal binary search tree, given the keys and their access probabilities. It is widely used in several fields, for example in the word prediction, which is the problem of guessing the next word in a sentence as the sentence is being entered, and updates this prediction as the word is typed [9]. The classical sequential algorithm for this problem requires $\mathcal{O}(n^3)$ calculation time and $\mathcal{O}(n^2)$ memory space [13].

1.1 Literature Review

Using the monotonicity property of optimal binary search trees (that we will define below), Knuth [22] derived an $\mathcal{O}(n^2)$ algorithm in the same space. Yao [34] proposed an algorithm with the same complexity thanks to the quadrangle inequalities. Using the restrictive assumption of convexity, Eppstein et al. [10] obtained an algorithm which required $\mathcal{O}(n \log n)$ time. During the resolution of the OBST problem, the sequential nature of the evaluation (diagonal-by-diagonal processing) of the dynamic programming table prevents from its efficient parallelization in the case of the PRAM model [2]. In the case of using real target parallel machines (systolic model for example), where the communication operations are taken into account, the parallel evaluation of this table becomes more complicated, because of the strong dependence relations between the subproblems (the evaluation of an element of the level d requires the already calculated values of two elements of levels $0, 1, \dots, d-1$), see [14, 18]. Although the parallelization of the classical algorithm has been widely studied by the community of researchers in parallel processing for different parallel computing models [4, 28, 29], little work is produced for the parallelization of the Knuth's approach. It is difficult to effectively parallelize this version on an abstract model. In PRAM model, it is unknown how to use the monotonicity property to reduce the number of processors in polylogarithmic calculations time [16, 17]. The solution becomes even more difficult to design, if we want to search the OBST solutions on a bridging model as the bulk synchronous parallel model (BSP for short) [32, 33]. Precisely, the goal is to benefit from the characteristics of a large majority of current parallel machines (where local memory size $\gg 1$) in order to minimize the communication overhead by regrouping them in global communication rounds between pure calculation super-steps, which is the objective of the CGM model.

In this paper, our aim is to parallelize Knuth's sequential algorithm for OBST problem on the bridging coarse grain

BSP/CGM (bulk synchronous parallel model/coarse-grained multicomputer) [5–8, 32]. CGM seems to be the best model for the design of algorithms that are not too dependent on a particular architecture. A BSP/CGM machine is a set of p processors, each having its own local memory of size s (with $\mathcal{O}(s) \gg \mathcal{O}(1)$) and connected to each other through a router able to deliver messages in a point-to-point manner. Each BSP/CGM parallel algorithm is an alternation of local computations and global communication rounds. Each communication round consists in routing a single h -relation with $h = \mathcal{O}(s)$. Each CGM computation or communication round corresponds to a BSP super-step having a communication cost $g \times s$ [4]. Here, g is the cost of a communication of a word in the BSP model. In order to produce an efficient BSP/CGM parallel algorithm, the effort of the designers must be to maximize the speed-up and minimize the number of communication rounds. (Ideally, it must be independent from the problem size and constant in the optimum.)

There are many CGM-based parallel algorithms for some dynamic programming problems, among others: The string editing problem [1], the longest common subsequence problem [2, 3, 12, 28]. The special class of problems including matrix chain ordering problem (MCOP), OBST and optimal string parenthesizing problem (OSPP) which are solved by the same dynamic programming algorithm, each with its own specification, was tackled in [15, 19, 26, 30].

In [20], the authors proposed a CGM-based parallel algorithm for MCOP that requires $\mathcal{O}\left(\frac{n^3}{p}\right)$ time steps and $\mathcal{O}(p)$ communication rounds on p processors. The one for OSPP in [30] requires $\lceil \sqrt{2p} \rceil$ communication rounds and $\mathcal{O}\left(\frac{n^3}{p}\right)$ execution time on p processors. Similarly, their algorithm for OBST in [27] requires $\lceil \sqrt{2p} \rceil$ communication rounds and $\mathcal{O}\left(\frac{n^2}{p}\right)$ time steps. A major drawback of these algorithms is that the loads of processors are unbalanced and they promote the idleness of processors.

1.2 Our Contribution

Dilson and Marco [15] have proposed a CGM-based parallel algorithm for MCOP which requires $\mathcal{O}(1)$ communication rounds and $\mathcal{O}\left(\frac{n^3}{p^3}\right)$ time steps. Their solution is based on the graph model proposed in [4], called dynamic graph. In fact, Bradford showed that at each instance of this problem it corresponds to a one-to-all shortest path problem in this graph. Thus, their algorithm is based on the computation of shortest paths in a dynamic graph derived from the input. It would be interesting to use the properties of such type of graph for solving the OBST problem.

Tchendji and Myoupo [27, 30] have shown that for OBST problem, load balancing of the processors and minimization of the communication time are two contradictory objectives when the corresponding tasks graph is partitioned into subgraphs (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 the minimization of communication rounds, the subgraphs must be of large size. Thus, the number of blocks of the task graph is minimized, and therefore, the number of communication rounds of the corresponding algorithm is reduced;
2. to promote load balancing of the processors (and therefore increase the efficiency of the parallel algorithm), the subgraphs must be of smaller size. Thus, if a processor has one more block to evaluate than another, because blocks are of small size, their load difference will also be low.

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) [31]. Their idea is based on a task graph partitioning technique and data distribution introduced in [27, 30]. In these approaches, the task graph is partitioned in the same way (in rows and columns of blocks), and data dependencies are the same. The steps of the algorithm based on user parameters are then deduced. The performances (complexity) of the algorithm are expressed from these parameters.

The main drawback of this solution is the contradictory optimization criterion caused by the regular size of blocks of the task 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 main contribution can be summarized as follows:

- firstly, we propose a dynamic graph model for solving the OBST problem and show that each instance of this problem corresponds to a one-to-all shortest path problem in this graph. So like in [15], any solution of a one-to-all shortest path problem can be used on our dynamic graph to accelerate the sequential or parallel resolution of OBST problem;
- secondly, we proposed a BSP/CGM parallel algorithm based on our dynamic graph to solve the OBST problem

Fig. 1 Sample of a binary search tree corresponding to the set of letter $\{c, e, f, g, h, k, l, n, o, r, s\}$ sorted according to the alphabetic order

from the corresponding one-to-all shortest path in this graph. It uses our new technique of irregular partitioning of this 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. This new technique assures that the blocks of the first diagonals (or the first steps) are of large sizes to minimize the number of communications, but this size decreases along the diagonals to increase the number of blocks in these diagonals and 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 follows: In Sect. 2, we give the definition of the OBST problem and a short review of its sequential solution. Section 3 presents our dynamic graph model for OBST problem. In Sect. 4, we present our BSP/CGM algorithm. The experimental results are analyzed in Sect. 5. Discussion about the constraints of the parallelization of the Knuth's sequential algorithm for the OBST problem is presented in the conclusion.

2 Optimal Binary Search Tree Problem

2.1 Classical Sequential Algorithm: Computing Optimal Solution in $\mathcal{O}(n^3)$

A binary search tree is a binary tree that serves to efficiently look for an element among a sorted set of elements (often called keys). Value and all the keys of the left sub-tree are less than the key at the root, and all the keys of the right sub-tree are greater than the key at the root. This property is applied recursively for the left and right sub-trees of this tree. A binary search tree completely orders its keys and is used to efficiently look for a particular key among a sorted set of keys.

The search cost of a particular key in a binary search tree depends on its depth in this tree. The example of Fig. 1 shows that the search cost of the key “n” is 1 and that of the key “l”

Fig. 2 For a sequence $\{a, b, c\}$, there are five possible binary search trees and the search cost of “a” is 1 on the first tree and 3 on the fourth

is 5. Thus, to order a set of n keys, $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ different binary trees are possible [21]. For a sequence $\{a, b, c\}$, Fig. 2 shows the different possible trees.

Therefore, given a set of keys and their access probabilities, the *optimal binary search tree* problem is to search the binary search tree for that set of keys which has the smallest cost. More formally, the problem can be defined as follows: consider that we have n keys $K_1 < K_2 < \dots < K_n$, which are to be placed in a binary search tree, and $2n + 1$ probabilities $p_1, p_2, \dots, p_n, q_0, q_1, q_2, \dots, q_n$ with $\sum p_i + \sum q_j = 1$ where p_i is the access probability of the key k_i and q_j is that of:

- a key which lies between k_j and k_{j+1} if $0 < j < n$;
- a key greater than k_n if $j = n$;
- a key less than k_1 if $j = 0$.

Let's denote T a binary search tree for this set of keys. Let b_i be the number of edges on the path from the root to the interior node k_i , and a_j be the number of edges on the path from the root to the leaf (k_j, k_{j+1}) . Then the expected cost induced by a search in T is given by: $cost(T) = \sum(p_i \times (b_i + 1)) + \sum(q_j \times a_j)$, since the access cost of key k_i , is $b_i + 1$, while for the gap (k_j, k_{j+1}) it is just a_j . An optimal binary search tree for this set of keys is a binary search tree whose expected search cost is smallest. From this, the optimal binary search tree problem boils down to construct an optimal binary search tree given a set of keys and their access probabilities.

Denote by $OBST(i, j)$ an optimal binary tree corresponding to the set of keys in the interval $int(i, j) = [K_{i+1}, \dots, K_j]$ and denote by $Tree(i, j)$ the cost induced by this tree. Let $w(i, j) = p_{i+1} + \dots + p_j + q_i + q_{i+1} + \dots + q_j$, thus $w(0, n) = 1$ and $w(i, i) = q_i$. The search costs $Tree(i, j)$ of $OBST(i, j)$ obey to the following dynamic programming recurrences for $0 \leq i \leq j \leq n$:

$$\begin{aligned} Tree(i, j) &= q_i & \text{if } i = j \\ Tree(i, j) &= w(i, j) + \min_{i \leq k < j} \{Tree(i, k) + Tree(k + 1, j)\} & \text{if } i \leq k < j \end{aligned} \quad (1)$$

The corresponding sequential generic algorithm for this class of problems requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space [13].

Fig. 3 Dynamic programming table or tasks graph for $n = 7$

Fig. 4 Diagonal-after-diagonal evolution of calculation of the dynamic programming table corresponding to the OBST problem (from the clearest to the darkest diagonal)

The cost values $Tree(i, j)$ are calculated according to their tabulation in an array, often called *dynamic programming table* or *shortest path matrix* denoted here by $Tree(0, n)$. In this table, each entry (i, j) corresponds to a subproblem and contains the value $Tree(i, j)$ at the end of the algorithm. For example, having seven keys ($n = 7$), the corresponding dynamic programming table $Tree(0, 7)$ (where (i, j) stands for $Tree(i, j)$) is represented in Fig. 3.

The bottom-up approach of this algorithm evaluates the entries of this table diagonal-by-diagonal, starting from the central diagonal (Fig. 4). The general structure of this algorithm is presented by Algorithm 1.

Algorithm 1: Classical algorithm**Data:** The $2n + 1$ probabilities.**Result:** The cost of the best binary search tree.

```

1 In diagonal 0, initialize each element  $(i, i)$  to  $q_i$  and all other
  elements to 0;
2 for each diagonal  $d$  from 1 to  $n$  do
3   for each element  $Tree(i, j)$  in diagonal  $d$  do
4      $Tree(i, j) \leftarrow$ 
        $w(i, j) + \min\{Tree(i, k) + Tree(k + 1, j) : i \leq k < j\};$ 
5      $Cut(i, j) \leftarrow$  the value of  $k$  that minimizes  $Tree(i, j)$ ;

```

When all entries of table $Tree(0, n)$ are evaluated (corresponding to the first step, or step of calculation or search of the optimal solution), a recursive algorithm with a time complexity $\mathcal{O}(n^2)$ is executed for the construction of the tree (corresponding to the second step or step of construction of the optimal solution). This recursive algorithm uses a table denoted $Cut(0, n)$, which is obtained from the table $Tree(0, n)$. Each entry (i, j) of this table gives the value of k (in Eq. 1) that minimizes $Tree(i, j)$. It is the first step that gives an optimal decomposition of the tree $OBST(i, j)$ in two sub-trees. In this paper, we are only interested by the parallelization of the first step of the resolution of this problem, i.e., the search of the optimal cost.

Proposed by Knuth, the fastest sequential algorithm requires $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space [22]. The acceleration obtained with this algorithm comes from the *property of monotonicity* verified by some entries of the matrix $Cut(0, n)$ [22].

In Sect. 2.2, we discuss how this acceleration can be used to derive an efficient BSP/CGM parallel algorithm for OBST. We present the Knuth's sequential algorithm which is based on our CGM algorithm.

2.2 Knuth-Acceleration-Based Sequential Algorithm in $\mathcal{O}(n^2)$

The Knuth's acceleration results from his observation of a property of the matrix $Cut(0, n)$ [22]. This property is defined as follows:

$$i \leq i' \leq j \leq j' \rightarrow Cut(i, j) \leq Cut(i', j')$$

This property transforms Algorithm 1 to the following (Algorithm 2).

Algorithm 2: Knuth sequential algorithm.**Data:** The $2n + 1$ probabilities.**Result:** The cost of the best binary search tree.

```

1 In diagonal 0, initialize each element  $(i, i)$  to  $q_i$  and all other
  elements to 0;
2 for each diagonal  $d$  from 1 to  $n$  do
3   for each element  $Tree(i, j)$  in diagonal  $d$  do
4      $Tree(i, j) \leftarrow w(i, j) + \min\{Tree(i, k) + Tree(k +$ 
        $1, j) : Cut(i, k - 1) \leq k < Cut(k + 1, j)\};$ 
5      $Cut(i, j) \leftarrow$  the value of  $k$  that minimizes  $Tree(i, j)$ ;

```

We can see from the Algorithm 2 that the acceleration comes from the internal for loop which requires only $\mathcal{O}(n)$ comparison operations unlike $\mathcal{O}(n^2)$ for the classical version. For the calculation of an entry (i, j) of diagonal $(j - i + 1)$, the number of comparison operations is reduced from $(j - i)$ to $(Cut(i + 1, j) - Cut(i, j - 1))$. The Knuth acceleration is not regular on all entries in the same diagonal. Indeed, the number of comparison operations defers from an entry to another. Thus, contrary to the classical version, if the entries of a diagonal are fairly distributed on different processors, nothing guarantees that these processors will have the same load. From this, a new constraint must be taken into account during the design of parallel algorithms based on Knuth's algorithm: it is "the inequality of necessary calculation loads for entries of the same diagonal of the dynamic programming table".

This property of Knuth's algorithm induces a load unbalance that is difficult to manage, and can vanish the acceleration obtained. One possible solution to inherit this acceleration and eliminate the global unbalanced loads is to evaluate each diagonal of the task graph by a single processor. But this is contrary to the main source of the parallelism of this algorithm, as for the classic $\mathcal{O}(n^3)$ version: the independence of computations between the entries of the same diagonal in the dynamic programming table.

In the next Sect. 3, we propose a dynamic graph model and show that the OBST problem can be solved through a one-to-all shortest paths problem in this graph.

3 Dynamic Graph Model for OBST Problem

3.1 Fundamental Concepts

In this section, we present the basic concepts for the dynamic graph model proposed in this paper.

Definition 1 A finite semigroupoid $(S, R, \bullet)$ is a non-empty finite set S , a binary relation $R \subseteq S \times S$, and an associative binary operator $\bullet$ satisfying the following conditions [24]:

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 $a_0 \bullet a_1 \bullet \dots \bullet a_n$ such that $(a_i, a_{i+1}) \in R$, $0 \leq i < n$. A linear product is a product of the form $((\dots (a_1 \bullet a_2) \bullet \dots) \bullet a_n)$ or $(a_1 \bullet (\dots (a_{n-1} \bullet a_n) \bullet \dots))$.

Definition 3 A weighted semigroupoid $(S, R, \bullet, pc)$ is a semigroupoid $(S, R, \bullet)$ with a nonnegative product cost

function pc such that if $(a_i, a_j) \in R$ then $pc(a_i, a_j)$ is the cost of evaluating $a_i \cdot a_j$. The minimal cost of evaluating an association product $a_{i+1} \cdot a_{i+2} \cdots a_j$ is denoted by $sp(i, j)$ and is defined by:

$$\begin{aligned} sp(i, j) &= Init(i) && \text{if } i = j \\ sp(i, j) &= \min_{i \leq k < j} \{sp(i, k) + sp(k+1, j) \\ &\quad + pc(a_{i+1} \cdots a_k, a_{k+1} \cdots a_j)\} && \text{if } i \leq k < j \end{aligned} \quad (2)$$

It is clear that when $pc(a_{i+1} \cdots a_k, a_{k+1} \cdots a_j) = w(i, j)$, $0 \leq i \leq k < j \leq n$ and $Init(i) = q_i$, $i \in \{0, 1, \dots, n\}$, Eq. 2 is equivalent to Eq. 1 in Sect. 2.1. So, we can derive the next corollary.

Corollary 1 *Given a weighted semigroupoid and an associative product $a_1 \cdot a_2 \cdots a_n$, when $pc(a_{i+1} \cdots a_k, a_{k+1} \cdots a_j) = w(i, j)$, $0 \leq i \leq k < j \leq n$ and $Init(i) = q_i$, $i \in \{0, 1, \dots, n\}$, find $sp(0, n)$ is to solve the OBST problem.*

3.2 Description of the Model

We designate by $K = (K_1, K_2, \dots, K_n)$ a sequence of n distinct keys sorted (so that $K_1 < K_2 < \dots < K_n$). We want to build an optimal binary search tree from these keys. Some searches may concern values that do not belong to K , so we also have $n+1$ “dummy keys” $d_0, d_1, \dots, d_n$ representing values outside K . For each key K_i (respectively dummy key d_i), we have the probability p_i (respectively q_i) that a search concerns K_i (respectively d_i). Each key K_i is an internal node, and each dummy key d_i is a leaf. The formalization of the OBST problem is carried out in Sect. 2.1.

We reference each vertex of our dynamic graph D_n by the pair of integers (i, j) , $0 \leq i \leq j \leq n$. We say that 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 contains $n+1$ 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 $sp(i, j)$. This is shown in Fig. 5a.

We designate the unit arcs of the dynamic graph D_n by $\rightarrow$, $\uparrow$ or $\nearrow$. The unit arc $(i, j) \rightarrow (i, j+1)$ represents associative product $(K_{i+1} \cdots K_j) \cdot K_{j+1}$, which is a binary sub-tree of root K_{j+1} whose left part contains the keys $K_{i+1} \dots K_j$ and dummy keys $d_i \dots d_j$ and its right part contains the dummy key d_{j+1} and weights $q_{j+1} + w(i, j+1)$. Similarly, the unit arc $(i, j) \uparrow (i-1, j)$ represents product $K_i \cdot (K_{i+1} \cdots K_j)$, corresponding to a binary sub-tree of root K_i whose left part contains the dummy key d_{i-1} and its right part contains the keys $K_{i+1} \dots K_j$ and dummy keys $d_i \dots d_j$ and weights

Fig. 5 Dynamic graphs D_3 and D'_3 for a set of $n = 3$ keys

$q_{i-1} + w(i-1, j)$. Also, for all i , $0 \leq i \leq n$, the arrows $\nearrow$ represent units arcs from $(-1, -1)$ to (i, i) and weights q_i .

To represent the split tree,¹ we add edges to D_n called jumpers. Denote a horizontal jumper by $\Rightarrow$ and a vertical jumper by $\Uparrow$. The vertical jumper $(i, j) \Uparrow (s, j)$ is $(i-s)$ units long and the horizontal jumper $(i, j) \Rightarrow (i, t)$ is $(t-j)$ units long, where all non-jumper edges are of length 1. The shortest path to the node (i, j) through the jumps $(i, k) \Rightarrow (i, j)$ and $(k+1, j) \Uparrow (i, j)$, represented by the product $(K_{i+1} \cdots K_k) \cdot (K_{k+1} \cdots K_j)$, defines the same binary sub-tree of root K_{k+1} whose left part contains the keys $K_{i+1} \dots K_k$ and dummy keys $d_i \dots d_k$, and its right part contains the keys $K_{k+2} \dots K_j$ and dummy keys $d_{k+1} \dots d_j$.

Definition 4 Given a set of n 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 arcs,

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

a set of jumpers,

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

a weight function W such that

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

¹ A tree splits into two sub-trees.

The dynamic graph D_3 , corresponding to an OBST problem for a set of $n = 3$ keys, is represented in Fig. 5a.

Let us note the similarity between a dynamic graph D_n and a conventional dynamic programming table, *Tree*, for the resolution of the OBST problem. The value of $sp(i, j)$ in the dynamic graph D_n is identical to $Tree(i, j)$ in the dynamic programming table.

Calculating a shortest path from $(-1, -1)$ to (i, j) gives the minimum cost of finding the OBST represented by the product $K_{i+1} \cdot K_i \cdot \dots \cdot K_j$, $0 \leq i < j \leq n$. So finding a shortest path from (i, j) to $(0, n)$ gives the minimum cost of finding the OBST represented by the product $K_1 \cdot \dots \cdot K_i \cdot P \cdot K_{j+1} \cdot \dots \cdot K_n$ where $P = (K_{i+1} \cdot \dots \cdot K_j)$.

Lemma 1 For all vertices (i, k) in a D_n graph, $sp(i, k)$ 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,

$$\begin{aligned} sp(i, k) &= sp(i, j) + W((i, j) \Rightarrow (i, k)) \\ &= sp(i, j) + sp(j + 1, k) + w(i, k) \end{aligned}$$

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 jumper $(j + 1, k) \Uparrow (i, k)$ is of length $j + 1 - i$. Therefore, since $j + 1 - i + k - j = k - i + 1$ and $k - j > \lceil (k - i)/2 \rceil$, we know that $j + 1 - i \leq \lceil (k - i)/2 \rceil$. On the other hand, a shortest path to $(j + 1, k)$ cannot contain a jumper longer than $k - (j + 1)$. Since $k - (j + 1) < k - j$, this Lemma follows inductively. $\square$

The Proof of Lemma 1 leads directly to Theorem 1.

Theorem 1 (Duality Theorem) If a shortest path from $(-1, -1)$ to (i, j) contains the jumper $(i, k) \Rightarrow (i, j)$, then there is a dual shortest path containing the jumper $(k + 1, j) \Uparrow (i, j)$.

Theorem 1 is fundamental because it makes it possible to avoid calculation redundancy when looking for the value of the shortest path for the vertices of the graph D_n . Indeed, for any vertex (i, j) , among all its shortest paths containing jumps, only those that contain only horizontal jumps are evaluated. The input graph of our BSP/CGM algorithm is therefore a subgraph of D_n denoted by D'_n , in which the set A' of the arcs consists of the set of unit arcs of $D_n \cup \{(i, j) \Rightarrow (i, t) : 0 \leq i < j < t \leq n\}$. Thus, in the graph D'_n , a vertex (i, j) has jumps only to the next vertices on the same line. Figure 5b shows the dynamic graph D'_3 .

Theorem 2 For each instance of OBST problem of a set of n elements, corresponds to a dynamic graph D_n , where $sp(i, j)$

is equal to $Tree(i, j)$. Thus, 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 a D_n graph corresponds to the cost of construction of a tree of an n element associative product. In addition, for every tree of an n element 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 a D_n graph, there is a corresponding associative product of n elements (or $n - 1$ operators “ $\cdot$ ”-s”). This proof is by induction on the lengths of the associative products. Consider D_4 and the product $K_1 \cdot K_2 \cdot K_3 \cdot K_4$. In D_4 , the jumper $(0, 2) \Rightarrow (0, 4)$ has weight $sp(3, 4) + w(0, 4)$. Therefore, the cost of a path using this jumper corresponds to the cost of the associative product $(K_1 \cdot K_2) \cdot (K_3 \cdot K_4)$. A symmetric argument holds for the jumper $(3, 4) \Uparrow (0, 4)$. 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, we only consider horizontal jumpers. For $m = 2k$ take a path in D_m from $(-1, -1)$ to $(0, m)$. We will show that for all $m \leq 2k$ there is a corresponding m element 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 $K_{i+1} \cdot \dots \cdot K_j$ by the inductive hypothesis. The product $K_{i+1} \cdot \dots \cdot K_j$ contains at most $k - 1$ associative operators (“ $\cdot$ ”-s”);
2. the cost of a path from (i, t) to $(1, 2k) = (0, m)$, $(1, 2k) = (0, m)$, where $k \leq t - i < m$, corresponds to the cost of a binary search tree of $K_1 \cdot \dots \cdot K_i \cdot P \cdot K_{t+1} \cdot \dots \cdot K_m$ where $P = (K_{i+1} \cdot \dots \cdot K_t)$, by the inductive hypothesis. Since P consists of at least k elements, the product $K_1 \cdot \dots \cdot K_i \cdot P \cdot K_{t+1} \cdot \dots \cdot K_m$ consists of at most $k - 1$ associative operators (“ $\cdot$ ”-s”).

Suppose that path from $(-1, -1)$ to $(1, 2k)$ includes the jumper $(i, j) \Rightarrow (i, t)$. 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 jumper $(i, j) \Rightarrow (i, t)$, where $j - i < k \leq t - i$. By the inductive hypothesis and fact 2, the cost of a path from (i, t) to $(0, m)$ corresponds to the cost of a binary search tree of $K_1 \cdot \dots \cdot K_i \cdot P \cdot K_{i+1} \cdot \dots \cdot K_m$ where $P = (K_{i+1} \cdot \dots \cdot K_i)$. Again by the inductive hypothesis and fact 1 above, the cost of a path to (i, j) corresponds to the cost of a binary search tree of $K_{i+1} \cdot \dots \cdot K_j$, where $K_i \cdot \dots \cdot K_j$ is a sub-product of P . Furthermore, the jumper $(i, j) \Rightarrow (i, t)$ is a part of a path from $(-1, -1)$ to $(0, m)$ whose cost corresponds to the cost of the product $(K_{i+1} \cdot \dots \cdot K_j) \cdot (K_{j+1} \cdot \dots \cdot K_t)$. This is because its cost is $sp(j+1, t) + w(i, t)$ and because $t - (j+i) < j - i$, we can apply the inductive hypothesis. Vertical jumpers follow similarly. $\square$

It is easy to derive the Corollary 2.

Corollary 2 *Any solution of a one-to-all shortest path problem can be used on our dynamic graph D_n to solve OBST problem.*

In the next section, we use our dynamic graph D_n as task graph for construction of our CGM solution.

4 CGM Algorithm for OBST Problem

This section presents CGM algorithm for OBST problem which construct the one-to-all shortest path in the dynamic graph D_n , on a BSP/CGM model of p processors, each with a local memory of size $\mathcal{O}(n^2/p)$. To reach this goal, firstly we partition the shortest path matrix (or the D_n graph) into sub-matrices (or subgraphs) of varying sizes (irregular size) in Sect. 4.1. Next, we study the dependencies between sub-problems in Sect. 4.2 and distribute the blocks onto processors in Sect. 4.3. Finally, we present the CGM algorithm in Sect. 4.4. This algorithm tries to reconcile the two contradictory objectives mentioned in Sect. 1.2.

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 load 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 $f(p) = \lceil \sqrt{2p} \rceil$, $g(n, p) = \lceil \frac{n+1}{f(p)} \rceil$ and $g(n, p, k) = \lceil \frac{g(n, p)}{2^k} \rceil$, we partition the shortest path matrix into sub-matrices or blocks ($Bk(i, j)$ for short). $Bk(i, j)$ is a $g(n, p, k) \times g(n, p, k)$ matrix at the k th fragmentation, i.e., at each fragmentation we divide the current size of the blocks into 4. Figure 6 shows two scenarios of this partitioning for $n = 31$, $k = 2$ and $p \in \{2, 3, 4\}$. The number in each block represents the diagonal in which it belongs.

In Fig. 6a, we have two diagonals of blocks when no fragmentation is performed. Since the number of blocks of the second diagonal become equal to the half of the first, we perform the first fragmentation on the block of second diagonal (this block of 16×16 size is divided into 4 blocks of 8×8 size) and the number of diagonals of blocks increases to 2. We perform the second fragmentation on the block of fourth diagonal when the number of blocks of this diagonal become equal to the half of the first (this block of 8×8 size is divided into 4 blocks of 4×4 size) and the number of diagonals of blocks increases to 2. Finally, we obtain a task graph with six diagonals of blocks.

It is important to note the following fundamental Remarks:

Remark 1

1. The blocks of the first diagonal are upper triangular matrices of $g(n, p)$ lines and $g(n, p)$ columns;
2. a block is full if it is a non-triangular matrix of size $g(n, p, k) \times g(n, p, k)$;
3. in general, blocks in the last column of blocks are not full (this is shown in Fig. 6b);
4. one fragmentation increases $\lceil \frac{f(p)}{2} \rceil + 1$ diagonals (see Proof of Lemma 3);
5. when $f(p)$ is odd, the number of blocks in a diagonal after each fragmentation exceeds by one notch the number of super-blocks (the larger blocks) of the first diagonal. (This is illustrated in Fig. 6b where there are 3 blocks in diagonal 1 and 4 blocks in the diagonal 3.)

This allows us to state the following Lemmas 2 and 3:

Lemma 2 *The number of blocks of the dynamic graph (the shortest path matrix) after partitioning is a function of k and is:*

(a) For $n = 31$, $k = 2$ and $p = 2$, the shortest path matrix is partitioned into nine blocks.

(b) For $n = 31$, $k = 2$ and $p \in \{3, 4\}$, the shortest path matrix is partitioned into twenty four blocks.

Fig. 6 Partitioning for $n = 31$, $k = 2$ and $p \in \{2, 3, 4\}$

$$C = (k-1) \times \frac{(S+1)(S+2) - \left\lceil \frac{S}{2} \right\rceil \left(\left\lceil \frac{S}{2} \right\rceil - 1 \right)}{2} + (S+1)^2 + \left\lceil \frac{S}{2} \right\rceil \frac{1 - \left\lceil \frac{S}{2} \right\rceil}{2} \quad \text{if } S \text{ is odd}$$

$$C = \frac{S^2(4+3k) + S(6k+4)}{8} \quad \text{else}$$

with $S = f(p)$.

Proof After partitioning, there is exactly $S(S+1)/2 - \lceil S/2 \rceil$ ($\lceil S/2 \rceil + 1)/2$ super-blocks. 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 in Fig. 6b where $k = 2$). This number increases by $S(S+1)/2 + \lceil S/2 \rceil$ blocks after the k th fragmentation; therefore,

$$C = (k-1) \times \left(\frac{S(S+1) - \left\lceil \frac{S}{2} \right\rceil \left(\left\lceil \frac{S}{2} \right\rceil - 1 \right)}{2} \right) + \left\lceil \frac{S}{2} \right\rceil + \frac{S(S+1)}{2} + \frac{S(S+1) - \left\lceil \frac{S}{2} \right\rceil \left(\left\lceil \frac{S}{2} \right\rceil + 1 \right)}{2} = \frac{S^2(4+3k) + S(6k+4)}{8}$$

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). Thus, we have:

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

□

Lemma 3 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 successive 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 in Fig. 6b). 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. $\square$

4.2 Block Dependency

Figure 7a, b shows the dependency of $Bk(i, j)$, $i < j$, on the other blocks after k fragmentations.

The extremities of a block $Bk(i, j)$ are defined as follows (see Fig. 7):

- LUE_{ij} stands for leftmost upper entry of $Bk(i, j)$;
- RUE_{ij} stands for rightmost upper entry of $Bk(i, j)$;
- LLE_{ij} stands for leftmost lower entry of $Bk(i, j)$;
- RLE_{ij} stands for rightmost lower entry of $Bk(i, j)$.

The expressions of these entries are the following:

- $LUE_{ij} = Tree(i, j - g(n, p, k) + 1)$;
- $RUE_{ij} = Tree(i, j)$;
- $LLE_{ij} = Tree(i + g(n, p, k) - 1, j - g(n, p, k) + 1)$;
- $RLE_{ij} = Tree(i + g(n, p, k) - 1, j)$.

The following Lemma 4 is inspired from [19]:

Lemma 4 (block dependency) *The calculation of entries of the block $Bk(i, j)$ requires, in the worst case, values of two blocks delimited, respectively, by extremities (ABCD) and (EFGH), as:*

- $A = Tree(i, Cut(RUE_{i, j-g(n, p, k)}) - 1)$
- $B = Tree(i, Cut(i + g(n, p, k), j) - 1)$
- $C = Tree(i + g(n, p, k) - 1, Cut(RUE_{i, j-g(n, p, k)}) - 1)$
- $D = Tree(i + g(n, p, k) - 1, Cut(i + g(n, p, k), j) - 1)$
- $E = Tree(Cut(RUE_{i, j-g(n, p, k)}) + 1, j - g(n, p, k) + 1)$
- $F = Tree(Cut(RUE_{i, j-g(n, p, k)}) + 1, j)$
- $G = Tree(Cut(i + g(n, p, k), j) + 1, j - g(n, p, k) + 1)$
- $H = Tree(Cut(i + g(n, p, k), j) + 1, j)$

Due to the unbalanced load caused by the Knuth acceleration, the sizes of the rectangles ABCD and EFGH and input contents from the task graph differ from one block to another. They are different even for blocks belonging to the same diagonal. Thus, the calculation load required for the evaluation of the blocks cannot be deduced prior to the treatment. The dependent relation between blocks shows that those that are not on the same line or on the same column

(a) Computational dependency of $Bk(i, j)$.

(b) Extremities of $Bk(i, j)$.

Fig. 7 Block dependency

of blocks is independent. Consequently, they can be carried out in parallel.

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

4.3 Mapping Blocks Onto Processors

This mapping has several advantages:

- However, it has a drawback: due to the snake-like data distribution onto processors, communications are not minimized.

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

- the k th fragmentation; thus, $1 + 3(k - 1) + 4 = 3k + 2$ 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. $\square$

4.4 CGM Algorithm

The overall CGM Algorithm that is run by every processor to solve the OBST problem is as follows (Algorithm 3).

Algorithm 3: CGM Algorithm.

Result: The tables $Tree(0, n)$ and $Cut(0, n)$ calculated.

1. Compute local solutions of the blocks of diagonal d using algorithm 2 ;
2. Communicate the entries (of *Cut* and *Tree* tables) necessary for calculation of each block of diagonals $\{d+1, d+2, \dots, f(p) + k \times (\lceil f(p)/2 \rceil + 1)\}$;

We are then ready to state Theorem 3 that gives the performances of our CGM Algorithm for OBST:

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

Proof At the k th fragmentation, the Knuth sequential Algorithm used in the local computation phases of our CGM Algorithm requires $\mathcal{O}\left(\frac{(n+1)^2}{2^{2k} \times (2p)^{2/2}}\right) = \mathcal{O}\left(\frac{n^2}{4^k \times (2p)}\right)$ local computations for the evaluation of each block of a diagonal of blocks. So, from the Proof of Lemma 5, we have for each processor:

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

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

Remark 3 When $k = 0$, our CGM Algorithm reduces to the one in [27], with $\mathcal{O}\left(\frac{n^2}{p}\right)$ time steps per processor and $\lceil \sqrt{2p} \rceil$ communication rounds.

It is crucial to know the better number of fragmentations before the start of computations in order to increase as most as possible the efficiency of the Algorithm. To do this, we propose to use the cost model defined in [11] to observe the evolution of load balancing, efficiency and the number of steps of the Algorithm that will result and determine, according to the triplet (n, p, k) , the better number of fragmentations to perform.

5 Experimental Results

Here we present the results of the implementation of our BSP/CGM Algorithm for solving the OBST problem. We implemented this Algorithm on the cluster dolphin of the MATRICS platform of the University of Picardie Jules Verne [25] using 60 computation nodes (48 nodes called thin nodes are with 128 GB of RAM, and 12 named thick

nodes with 512 GB of RAM), and whose characteristics are as follows:

- each node is an Intel Xeon Processor E5-2680 V4 (35M Cache, 2.40 GHz);
- 2 login nodes which are not intended to make the calculations but to give the job submission environment;
- 1 NFS server (85 TB) in 10 Gbits;
- a distributed file system BeeGFS (300 TB) in 100 Gbits.

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 (OpenMPI 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 {511, 1023, 2047, 4095, 8191, 16,383, 32,767};
- p is the number of processors, with values in the set {1, 2, 5, 8, 25, 28, 32, 114, 120, 128};
- k is the number of fragmentations of blocks performed, with values in the set {0, 1, 2}.

Recall that the communication time considered here is the sum of times of effective transfer of data and waiting times of the processors.

We consider that the parallel execution time T_{par} of the algorithm is the time interval between when the first processor starts its computations and when the last processor finishes its own. Its speed-up S_{par} will be evaluated by comparing T_{par} and the sequential execution time T_{seq} for the same problem; therefore, $S_{\text{par}} = \frac{T_{\text{seq}}}{T_{\text{par}}}$. Its efficiency E_{par} measures the average cost of processor activity and will be evaluated by $E_{\text{par}} = \frac{S_{\text{par}}}{p}$.

Figure 9a shows that communication time decreases as the number of fragmentations increases (on 32 processors, it decreases on average by 18.83% for $k = 1$ and on average by 25.08% for $k = 2$) because when a fragmentation is performed on the diagonal blocks, it increases the number of blocks in these diagonals and allows processors to stay active as long as possible. This minimizes the latency of processors and thus minimizes the communication time. Figure 9b shows that the computation rate is much lower than the communication rate. This is due to the computation time gain from the Knuth acceleration. The proportion of calculation is much lower with $k = 2$ than that of $k = 0$ (on 32 processors, it decreases on average by 50.19% for $k = 1$ and on average by 66.27% for $k = 2$) because each fragmentation divides the current size of the blocks into 4. This reduction in the size of blocks reduces greatly the number of

(a) Global communication time in function of p with $p \in \{2, 5, 8, 25, 28, 32, 114, 120, 128\}$, $n = 16383$ and $k \in \{0, 1, 2\}$.

(b) Computation rate versus communication rate for $p = 32$, $n \in \{511, 1023, \dots, 16383, 32767\}$ and $k \in \{0, 1, 2\}$.

Fig. 9 Global communication time and computation rate versus communication rate

local computations performed by a processor due to the load balancing and thus reduces the overall computation time.

Table 1 and Fig. 10a, b show that the reduction in computation and communication time results in a reduction in the total execution time as the number of fragmentations increases. In Fig. 10a, on 32 processors, it decreases on average by 25.67% for $k = 1$ and on average by 33.99% for $k = 2$. Also, when $n = 16,383$ in Fig. 10b, it decreases on average by 30.87% for $k = 1$ and on average by 40.43% for $k = 2$. Since the total execution time decreases when the number of fragmentations increases, the speed-up and efficiency of our algorithm increase. On 32 processors when $n = 16,383$ in Fig. 10b and Table 2 (respectively Table 3), speed-up (efficiency) is equal on average by 2.5 (respectively 7.83%) for $k = 0$, and increases up to 3.68 (respectively 11.52%) for $k = 1$ and 4.31 (respectively 13.49%) for $k = 2$. From all this, we can deduce that our algorithm is scalable to the increase in data size and the number of processors.

Figure 11a presents for $k \in \{0, 1, 2\}$ the curves of different loads compared to their average load. (Each value of k has its own load average.) This figure shows that irregular

(a) Total execution time in function of p with $p \in \{2, 5, 8, 25, 28, 32, 114, 120, 128\}$, $n = 16383$ and $k \in \{0, 1, 2\}$.

(b) Total execution time in function of n with $p \in \{1, 32\}$, $n \in \{511, 1023, \dots, 16383, 32767\}$ and $k \in \{0, 1, 2\}$.

Fig. 10 Total execution time in function of p and n

partitioning of the dynamic graph balances the load between the processors better than regular partitioning of this graph. Note that it is not possible to predict which processor will have the biggest or the smaller load, because of the irregularity in the number of calculations needed to evaluate the elements of the same diagonal induced by Knuth acceleration. For example, in Fig. 11a, $p = 5$ (respectively $p = 6$) has the greatest burden with the irregular partitioning of the dynamic graph when $k = 1$ (respectively when $k = 2$) with respect to $p = 2$ (respectively $p = 3$) that should have been predicted to support this greatest burden.

In Fig. 11b, we can see that the load of more than half of the processors is lower than the average load (indeed, the loads of the two processors $p = 2$ and $p = 3$ are more or less equal; it is the same for processors $p = 5$ and $p = 6$). A processor loads below the average are very close to it ($p = 7$). Those that are above this average are those of the processors that evaluate the last blocks of the dynamic graph (task graph) and are more distant from the average load. This result was expected, because the load induced by a block of the task graph increases with the number of the diagonal on which it belongs. From this, we can

Table 1 Total execution time of our algorithm for $p \in \{1, 2, 8, 32, 128\}$, $n \in \{511, 1023, \dots, 16,383, 32,767\}$ and $k \in \{0, 1, 2\}$

p	n						
	511	1027	2047	4095	8191	16,383	32,767
$k = 0$							
1	0.23	1.05	6.11	39.67	258.79	2088.16	17,240.77
2	0.19	1.01	5.70	36.80	257.51	2054.92	15,728.34
8	0.15	0.77	4.32	27.47	190.33	1458.98	10,785
32	0.09	0.46	2.58	16.27	112.56	832.55	6365.66
128	0.05	0.25	1.39	8.76	60.21	443.98	3384.43
$k = 1$							
2	0.17	0.88	4.87	30.47	208.72	1549.23	12444.43
8	0.13	0.64	3.43	20.69	138.59	1040.62	7851.66
32	0.07	0.37	1.97	11.86	78.51	566.24	4258.75
128	0.04	0.21	1.09	6.39	41.72	296.80	2222.54
$k = 2$							
2	0.17	0.89	4.67	28.30	190.92	1392.15	10529.33
8	0.12	0.59	3.12	18.50	123.45	905.05	6798.14
32	0.07	0.34	1.78	10.40	67.79	483.59	3591.39
128	0.04	0.19	0.98	5.62	35.91	251.51	1870.06

Table 2 Speed-up of our algorithm for $p \in \{2, 8, 32, 128\}$, $n \in \{511, 1023, \dots, 16,383, 32,767\}$ and $k \in \{0, 1, 2\}$

p	n						
	511	1027	2047	4095	8191	16,383	32,767
$k = 0$							
2	1.20	1.04	1.07	1.07	1	1.01	1.09
8	1.50	1.36	1.41	1.44	1.35	1.43	1.59
32	2.53	2.27	2.36	2.43	2.29	2.50	2.70
128	4.55	4.14	4.36	4.52	4.29	4.70	5.09
$k = 1$							
2	1.31	1.19	1.25	1.30	1.23	1.34	1.38
8	1.75	1.64	1.77	1.91	1.86	2	2.19
32	2.98	2.80	3.09	3.34	3.29	3.68	4.04
128	5.02	4.94	5.60	6.20	6.19	7.03	7.75
$k = 2$							
2	1.34	1.18	1.30	1.40	1.35	1.49	1.63
8	1.89	1.77	1.95	2.14	2.09	2.30	2.53
32	3.16	3.06	3.43	3.81	3.81	4.31	4.80
128	5.23	5.32	6.21	7.05	7.19	8.30	9.21

conclude that our mapping (data distribution) promote load balancing of the processors.

6 Conclusion and Future Works

In this paper, we presented an efficient parallel algorithm on the BSP/CGM model for solving the OBST problem on p processors. Firstly, we proposed a dynamic graph model for solving the OBST problem and showed that each instance of this problem corresponds to a one-to-all shortest path problem in this graph. So like in [15], any solution

of a one-to-all shortest path problem can be used on our dynamic graph to accelerate the sequential or parallel resolution of OBST problem. Secondly, we proposed a BSP/CGM parallel algorithm based on our dynamic graph to solve the OBST problem from the corresponding one-to-all shortest path in this graph. It uses our new technique of irregular partitioning of this 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. In our BSP/CGM algorithm, each processor has $\mathcal{O}\left(\frac{n^2}{p}\right)$ local mem-

Table 3 Efficiency (%) of our algorithm for $p \in \{2, 8, 32, 128\}$, $n \in \{511, 1023, \dots, 16,383, 32,767\}$ and $k \in \{0, 1, 2\}$

p	n						
	511	1027	2047	4095	8191	16,383	32,767
$k = 0$							
2	60.29	52.30	53.61	53.89	50.19	50.80	54.80
8	18.78	17.05	17.64	18.04	16.97	17.89	19.98
32	7.92	7.10	7.39	7.61	7.17	7.83	8.46
128	3.56	3.23	3.41	3.53	3.35	3.67	3.97
$k = 1$							
2	65.87	59.80	62.69	65.08	61.92	67.39	69.27
8	21.99	20.54	22.21	23.96	23.31	25.08	27.44
32	9.31	8.75	9.67	10.44	10.28	11.52	12.65
128	3.92	3.86	4.38	4.84	4.84	5.49	6.06
$k = 2$							
2	67.05	59.10	65.30	70.07	67.69	74.99	81.87
8	23.68	22.16	24.41	26.80	26.17	28.84	31.70
32	9.88	9.57	10.72	11.91	11.91	13.49	15
128	4.09	4.16	4.85	5.51	5.62	6.48	7.20

(a) Load difference relative to the average load for $p = 8$, $n \in \{511, 1023, \dots, 16383, 32767\}$ and $k \in \{0, 1, 2\}$.

(b) Load difference relative to the average load for $p = 8$, $k = 2$ and $n \in \{511, 1023, \dots, 16383, 32767\}$.

Fig. 11 Load difference relative to the average load in function of p

ory. It runs, in the worst case, in $\mathcal{O}\left(\frac{n^2}{p}\right)$ time steps per processor and $\lceil \sqrt{2p} \rceil + k \times \left(\left\lceil \frac{\lceil \sqrt{2p} \rceil}{2} \right\rceil + 1\right)$ communication rounds. The experimental results show a good agreement with theoretical predictions. The progressive reduction in size of the blocks allows processors to stay active as long as possible. This promotes load balancing and thus minimizes the overall computation times of processors. This also reduces their latency and then minimizes the communication time. All this reduces the overall execution time of the algorithm.

In perspective of this work, it will be interesting to use our dynamic graph model to develop a CGM-based parallel algorithm for OBST problem which requires $\mathcal{O}(1)$ communication round and $\mathcal{O}\left(\frac{n^2}{p^2}\right)$ time steps, using, for example, the works of [15]. The irregular partitioning technique of the tasks graph may be applicable to other dynamic programming problems in the same class of the OBST as matrix chain ordering problem. This work is also left for future works.

Acknowledgements The authors wish to express their gratitude to the University of Picardie Jules Verne which made it possible to carry out the experimentations of this work.

Author's Contribution VK suggested this work. The authors carried out the analysis. JL performed the experiments and wrote the first draft of this work. The authors worked for the revised version and approved the work.

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Alves CER, Cáceres EN, Dehne F (2002) Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In: Proceedings of the fourteenth annual ACM symposium on parallel algorithms and architectures, ACM, New York, NY, USA, SPAA'02, pp 275–281
- Alves CER, Cáceres EN, Dehne F, Song SW (2003) A parallel wavefront algorithm for efficient biological sequence comparison. In: Proceedings of the 2003 international conference on Computational science and its applications: PartII, ICCSA'03, pp 249–258
- Alves CER, Cáceres EN, Song SW (2006) A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica* 45:301–335
- Bradford PG (1994) Parallel dynamic programming. Ph.D. thesis, Indiana University
- Cáceres E, Dehne F, Ferreira A, Flocchini P, Rieping I, Roncato A, Santoro N, Song S (1997) Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In: *Algorithmica*. Springer, pp 390–400
- Cheatham T, Fahmy A, Stefanescu D, Valiant L (1995) Bulk synchronous parallel computing—a paradigm for transportable software. In: Proceedings of the twenty-eighth annual hawaii international conference on system sciences, vol 2, pp 268–275
- Dehne F, Fabri A, Rau-chaplin A (1994) Scalable parallel computational geometry for coarse grained multicomputers. *Int J Comput Geom* 6:298–307
- Dehne F, Fabri A, Rau-Chaplin A (1996) Scalable parallel computational geometry for coarse grained multicomputers. *Int J Comput Geom Appl* 6(3):379–400
- El-Qawasmeh E (2004) Word prediction using a clustered optimal binary search tree. *Inf Process Lett* 92(5):257–265
- Eppstein D, Galil Z, Giancarlo R (1988) Speeding up dynamic programming. In: Proceedings of 29th symposium in foundations of computer science, pp 488–496
- Fotso LP, Tchendji VK, Myoupo JF (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, Las Vegas, USA, pp 710–716
- Garcia T, Myoupo J, Semé D (2003) A coarse-grained multicomputer algorithm for the longest common subsequence problem. In: Eleventh Euromicro conference on parallel, distributed and network-based processing, pp 349–356
- Godbole SS (1973) On efficient computation of matrix chain products. *IEEE Trans Comput* 100(9):864–866
- Guibas LJ, Kung HT, Thompson CD (1979) Direct VLSI implementation of combinatorial algorithms. In: Proceedings of the caltech conference on very large scale integration. California Institute of Technology, Pasadena, CA, USA, pp 509–525
- Higa DR, Stefanescu MA (2012) A coarse-grained parallel algorithm for the matrix chain order problem. In: Proceedings of the 2012 symposium on high performance computing. Society for Computer Simulation International, San Diego, CA, USA, HPC '12, pp 1–8
- Karpinski M, Rytter W (1994) On a sublinear time parallel construction of optimal binary search trees. In: MFCS, Springer, Lecture Notes in Computer Science, vol 841, pp 453–461
- Karpinski M, Larmore L, 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, Philadelphia, PA, USA, SODA'96, pp 36–41
- Karypis G, Kumar V (1993) Efficient parallel mappings of a dynamic programming algorithm: a summary of results. In: Proceedings seventh international parallel processing symposium, pp 563–568
- Kechid M, Myoupo J (2008) A coarse grain multicomputer algorithm solving the optimal binary search tree problem. In: Proceedings of the Fifth international conference on information technology: new generations. IEEE Computer Society, Washington, DC, USA, pp 1186–1189
- Kechid M, Myoupo J (2008) An efficient BSP/CGM algorithm for the matrix chain ordering problem. In: The 2008 international conference on parallel and distributed processing techniques and applications, Las Vegas, USA, pp 327–332
- Knuth D (1971) Optimum binary search trees. *Acta Inform* 1:14–25
- Knuth DE (1972) Optimum binary search trees. *Acta Inform* 1(3):270–270
- Li GJ, Wah BW (1985) Parallel processing of serial dynamic programming problems. *Proc COMPSAC* 85:81–89
- Marvins M (1978) Introduction to modern algebra. Marcel Dekker, New York
- Matrices Platform (2019). <https://www.u-picardie.fr/recherche/presentation/plateformes/plateforme-matrices-382844.kjsp>
- Myoupo JF, Tchendji VK (2014) An efficient CGM-based parallel algorithm solving the matrix chain ordering problem. *Int J Grid High Perform Comput (IJGHPC)* 6(2):74–100
- Myoupo JF, Tchendji VK (2014) Parallel dynamic programming for solving the optimal search binary tree problem on CGM. *Int J High Perform Comput Netw* 7(4):269–280
- Rytter W (1988) On efficient parallel computations for some dynamic programming problems. *Theor Comput Sci* 59:297–307
- Tang D, Gupta G (1995) An efficient parallel dynamic programming algorithm. *Comput Math Appl* 30(8):65–74
- Tchendji VK, Myoupo JF (2012) An efficient coarse-grain multicomputer algorithm for the minimum cost parenthesizing problem. *J Supercomput* 61:463–480
- Tchendji VK, Myoupo JF, Dequen G (2016) High performance CGM-based parallel algorithms for the optimal binary search tree problem. *Int J Grid High Perform Comput* 8(4):55–77
- Valiant L (1990) A bridging model for parallel computation. *Commun ACM* 33:103–111
- Valiant LG (1989) Bulk-synchronous parallel computers. In: *Parallel Processing and Artificial Intelligence*, pp. 15–22. John Wiley & Sons.
- Yao F (1982) Speed-up in dynamic programming. *SIAM J Matrix Anal Appl* 3(4):532–540