ARTICLE TEMPLATE

Four-splitting based coarse-grained multicomputer parallel algorithm for the optimal binary search tree problem

Jerry Lacmou Zeutouo[a], Vianney Kengne Tchendji[a], and Jean Frédéric Myoupo[b]

[a]Department of Mathematics and Computer Science, University of Dschang, Dschang, Cameroon; [b]Computer Science Lab-MIS, University of Picardie Jules Verne, Amiens, France

ABSTRACT

This paper presents a parallel solution based on the coarse-grained multicomputer (CGM) model using the four-splitting technique to solve the optimal binary search tree problem. The well-known sequential algorithm of Knuth solves this problem in $\mathcal{O}\left(n^2\right)$ time and space, where n is the number of keys used to build the optimal binary search tree. To parallelize this algorithm on the CGM model, the irregular partitioning technique, consisting in subdividing the dependency graph into subgraphs (or blocks) of variable size, has been proposed to tackle the trade-off of minimizing the number of communication rounds and balancing the load of processors. This technique however induces a high latency time of processors (which accounts for most of the global communication time) because varying the blocks' sizes does not enable them to start evaluating some blocks as soon as the data they need are available. The four-splitting technique proposed in this paper solves this shortcoming by evaluating a block as a sequence of computation and communication steps of four subblocks. This CGM-based parallel solution requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds, where p is the number of processors and k is the number of times the size of blocks is subdivided. An experimental study conducted to evaluate the performance of this CGM-based parallel solution showed that compared to the solution based on the irregular partitioning technique where the speedup factor is up to $\times 10.39$ on one hundred and twenty-eight processors with 40960 keys when $k = 2$, the speedup factor of this solution is up to $\times 13.12$ and rises up to $\times 14.93$ when $k = 5$.

1. Introduction

The optimal binary search tree problem consists in determining the binary search tree that will have the lowest overall cost of searching from a set of sorted keys knowing their access probabilities. It is a non-serial polyadic dynamic-programming problem which is characterized by a strong dependency between subproblems [1]. This problem is widely studied since applications of optimal search binary trees are numerous. One of the most obvious is the search for a word in a dictionary. Indeed, from the words of a dictionary and the access frequency of each word, an optimal binary search tree

---

CONTACT Vianney Kengne Tchendji. Email: vianneykengne@yahoo.fr

can be built to quickly and efficiently answer a query. An application derived from the latter is the translation of a word from one language to another [2]. El-Qawasmeh [3] used an optimal binary search tree to solve the word prediction problem. This problem attempts to guess and update the next word in a sentence as it is typed. In view of these different applications, an optimal binary search tree should be constructed from Knuth's sequential algorithm running in $\mathcal{O}\left(n^2\right)$ time and space [4]. It is an improved version of Godbole's sequential algorithm requiring $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space [5].

1.1. Related work

In the literature, researchers proposed several parallel solutions of Knuth's sequential algorithm on different parallel computing models. On the PRAM model, Karpinski et al. [6] designed a sublinear time parallel algorithm to construct optimal binary search trees. It requires $\mathcal{O}\left(n^{1-\epsilon}\log n\right)$ computation time with the total work $\mathcal{O}\left(n^{2+2\epsilon}\right)$ for an arbitrarily small constant $0 < \epsilon \leq 0.5$. Recall that the total work is equal to the computation time multiplied by the number of processors [6]. On CREW-PRAM machines, Karpinski et al. [7] presented an algorithm running in $\mathcal{O}\left(n^{0.6}\right)$ computation time with n processors. On realistic models of parallel machines, Craus [8] proposed an $\mathcal{O}\left(n\right)$-time parallel algorithm using $\mathcal{O}\left(n^2\right)$ processors on very-large-scale integration architectures. Wani and Admad [9] proposed a parallel implementation of Knuth's sequential algorithm on GPU architectures. With 16384 keys, they achieved a speedup factor of 409 on a NVIDIA GeForce GTX 570 chip and a speedup factor of 745 on a NVIDIA GeForce GTX 1060 chip. On shared-memory architectures, after demonstrating that dependencies of subproblems available in the code implementing the Knuth sequential algorithm enable to generate only 2D tiled code, using the polyhedral model, Bielecki et al. [10] proposed a way of transforming this algorithm to a modified one exposing dependencies to generate 3D parallel tiled code. Experimentations conducted using OpenMP demonstrated that the 3D tiled code considerably outperforms the 2D tiled code. On distributed-memory architectures, many researchers proposed their parallel solutions on the coarse-grained multicomputer (CGM) model.

The CGM model introduced in [11,12] is the most suitable to design parallel algorithms that are not too dependent on a specific architecture than the systolic and hypercube models. It enables to formalize in just two parameters the performance of a parallel algorithm : the input data size n and the number of processors p. A CGM-based parallel algorithm consists in successively repeating a computation round and a communication round until the problem is solved. Processors perform local computations on their data using the best sequential algorithm in each computation round and exchange data through the network in each communication round. All information sent from one processor to another is wrapped into a single long message to minimize the overall message overhead.

Kechid and Myoupo [13] proposed the first CGM-based parallel solution to solve the OBST problem in $\mathcal{O}\left(n^2/p\right)$ execution time with $\mathcal{O}\left(p\right)$ communication rounds. Myoupo and Kengne [14] built a CGM-based parallel solution running in $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(\sqrt{p}\right)$ communication rounds. Later on, Kengne et al. [15] noticed that the execution time was related to the load balancing and the number of communication rounds. These criteria depend on the partitioning strategy and the distribution scheme strategy used when designing the CGM-based parallel solution:

(1) When the dependency graph is subdivided into small-size subgraphs (or blocks) [13], the load difference between processors is small if one processor has one more

block than another. However, the number of communication rounds will be high.

(2) When the dependency graph is subdivided into large-size blocks [14], the number of communication rounds of the corresponding algorithm is reduced since there are few blocks. However, the load of processors will be unbalanced.

By generalizing the ideas of the dependency graph partitioning and distribution scheme introduced in [13,14], Kengne et al. [15] proposed a CGM-based parallel solution that gives to the end-user the choice to optimize one criterion according to his own goal. The main drawback of this solution is the conflicting optimization criteria owing to the fact that the end-user cannot optimize more than one criterion. Moreover, these criteria have a significant impact on the latency time of processors, which in turn has an impact on the global communication time. Recall that the global communication time is obtained by adding up the latency time of processors and the effective transfer time of data. Indeed, when the number of communication rounds is high, excessive communication will lead to communication overhead which will deteriorate the global communication time. On the other hand, when the load of processors is unbalanced, the evaluation of a block will take a long time because of its large size. So a processor that is waiting for this block to start or continue its computation will wait longer to receive this block. As a result, whatever the chosen criterion, the global communication time will decrease the performance of CGM-based parallel solutions.

To tackle this trade-off, Kengne and Lacmou [16] proposed the irregular partitioning technique of the dynamic graph. It consists in subdividing this graph into blocks of variable size. It ensures that the blocks of the first steps (or diagonals) are of large sizes to minimize the number of communication rounds. Thereafter, it decreases these sizes along the diagonals to increase the number of blocks in these diagonals and enable processors to stay active longer. These blocks are fairly distributed over processors to minimize their idle time and balance the load between them. It requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds, where k is the number of times the size of blocks is subdivided. Experimental results showed that the irregular partitioning technique significantly reduced the global communication time compared to the regular partitioning technique proposed in [13–15].

Nevertheless, this technique also induces a high latency time of processors since it does not enable processors to start evaluating small-size blocks as soon as the data they need are available. Yet, these data are usually available before the end of the evaluation of large-size blocks.

1.2. Our contribution

To solve the minimum cost parenthesizing problem (MPP), which is part of the same class of non-serial polyadic dynamic-programming problems than the OBST problem, Lacmou and Kengne [17] proposed the k-block splitting technique to reduce this latency time. This technique consists in splitting the large-size blocks into a set of smaller-size blocks called k-blocks. Thus, to enable processors to start the evaluation of k-blocks as soon as possible, a single processor evaluates a block by computing and communicating each k-block contained in this block.

The purpose of splitting the blocks into a set of k-blocks is to progressively evaluate and communicate them during the evaluation of blocks. Indeed, evaluating a block belonging to the diagonal d in a progressive fashion consists in starting at the diagonal $\lceil d/2 \rceil$. The k-block splitting technique is essentially based on the progressive evaluation of blocks. It is therefore adequate to solve the MPP because Godbole's sequential

3

algorithm gives the possibility to evaluate the nodes of blocks in this way. In contrast, Knuth's sequential algorithm does not enable to perform this kind of evaluation to solve the OBST problem because it does not enable to know in advance when to start or continue the evaluation of a node. Moreover, since the k-block are numerous and small when k increases, computing and communicating them to processors that need them will lead to communication overhead [17]. Thus, it would not be meaningful and practical to apply the k-block splitting technique to solve the OBST problem because a lot of unnecessary computations could be performed and lead to poor performance.

In this paper, we introduce the four-splitting technique to reduce the latency time of processors caused by the irregular partitioning technique by splitting the large-size blocks into four small-size blocks (or subblocks). Hence, evaluating a block by a single processor will consist of computing and communicating each subblock contained in this block. The goal is the same than the k-block splitting technique, that is, to enable processors to start the evaluation of blocks as soon as possible. This CGM-based parallel solution requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds. An experimental study conducted to evaluate the performance of this CGM-based parallel solution showed that compared to the solution based on the irregular partitioning technique where the speedup factor is up to $\times 10.39$ on one hundred and twenty-eight processors with 40960 keys when $k = 2$, the speedup factor of this new solution is up to $\times 13.12$ and increases up to $\times 14.93$ when $k = 5$. We also conducted experiments to determine the ideal number of times the size of the blocks is subdivided. The results showed that this number depends on the architecture where the solution is executed.

This paper is organized as follows: Section 2 describes the OBST problem and presents Knuth's sequential algorithm. Section 3 illustrates the dynamic graph model of the OBST problem introduced in [16]. Then, Section 4 presents our CGM-based parallel solution using the four-splitting technique. Section 5 outlines the experimental results obtained, and finally Section 6 concludes this work.

2. Optimal binary search tree problem

A binary search tree is a binary tree that maintains a set of sorted keys according to the following rule: for each vertex x of the tree, the key of x is larger than all the keys contained in the left subtree of x and it is smaller than all the keys contained in the right subtree of x [18]. Figure 1 shows an example of a binary search tree corresponding to the set of keys ⟨c, e, f, g, h, k, l, n, o, r, s⟩ sorted in alphabetical order. A binary search tree is typically used to efficiently search for a particular key among a set of sorted keys. The cost of searching for this key is equal to length of the path from the root to this key plus one, which is in fact, the depth of this key [2]. For example, in Figure 1, the cost of searching for the key "n" (the root vertex) is equal to 1 and the cost of searching for the key "h" is equal to 5.

The overall cost of searching, that is equal to the sum of the cost of searching of each key, must be usually as less as possible in many applications. Self-balancing binary search trees, which are binary search trees that automatically attempt to keep its height (maximal number of levels below the root vertex) as small as possible at all times [19], minimize this cost by ensuring that all keys are as near the root vertex as possible. However, when a query frequency is associated with each key, this tree will not be the most efficient in some cases. Indeed, it would be better to place the most frequently queried keys closer to the root vertex, and to place the less frequently

Figure 1. Example of a binary search tree corresponding to the set of letters ⟨c, e, f, g, h, k, l, n, o, r, s⟩ sorted in alphabetical order

Figure 2. Five possible binary search trees obtained from the set of sorted keys ⟨a, b, c⟩

queried keys further from the root vertex. For example, consider the set of sorted keys ⟨a, b, c⟩ with respective frequencies of 3, 1, and 7. The cost of searching for a key x is equal to the depth of x multiplied by the frequency of x. Of the five possible binary search trees, depicted in Figure 2, that can be obtained from these keys, the tree that minimizes the overall cost of searching is determined as follows:

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

As shown in Figure 2, the third binary search tree is balanced but the fifth is not. Nevertheless, the overall search cost of the latter is lower than the self-balanced binary search tree (the third tree in Figure 2) and is the lowest of all; thus, it is the optimal binary search tree for the above example.

More formally, the optimal binary search tree (OBST) problem can be defined as follows: consider a set of n sorted keys $K = \langle k_1, k_2, \ldots, k_n \rangle$, such that $k_1 < k_2 < \cdots < k_n$. The probability of finding a key k_i is denoted by p_i. In the case where the searched element is not in K, consider a set of $n+1$ dummy keys $D = \langle d_0, d_1, \ldots, d_n \rangle$. Indeed, d_0 represents the set of values that are smaller than k_1 and d_n the set of values that are larger than k_n. A dummy key d_i, such that $1 \leq i < n$, represents the set of

Algorithm 1 Godbole's sequential algorithm

1: **for** $i = 0$ to n **do**
2:     $Tree[i, i] \leftarrow q_i$;
3: **for** $d = 1$ to n **do**
4:     **for** $i = 0$ to $n - d + 1$ **do**
5:         $j \leftarrow n - d + 1$;
6:         $Tree[i, j] \leftarrow \infty$;
7:         **for** $k = i$ to $j - 1$ **do**
8:             $c \leftarrow Tree[i, k] + Tree[k + 1, j] + w(i, j)$;
9:             **if** $c < Tree[i, j]$ **then**
10:               $Tree[i, j] \leftarrow c$;
11:               $Cut[i, j] \leftarrow k$;
---

values between k_i and k_{i+1}. The probability of finding a dummy key d_i is denoted by q_i. In summary, either the search ends in success (i.e. finding the searched key k_i) or in failure (i.e. finding a dummy key d_i); thus the sum of probabilities of success and failure is equal to 1. A binary search tree T constructed from these different keys is composed of n internal vertices in the set K and $n + 1$ leaves in the set D. The overall cost of searching of T is given by Equation (1) :

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

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

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

$$Tree[i, j] = \begin{cases} q_i & \text{if } 0 \leq i = j \leq n, \\ \min_{i \leq k < j} \{Tree[i, k] + Tree[k + 1, j] + w(i, j)\} & \text{if } 0 \leq i < j \leq n. \end{cases} \tag{2}$$

Finding the optimal binary search tree from the set of keys K and the set of dummy keys D is reduced to compute $Tree[0, n]$. The straightforward sequential algorithm of Godboble [5], running in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, is given by Algorithm 1. The dynamic-programming (DP) table (named $Tree$ in Algorithm 1) stores the value of the optimal cost of searching of $T_{i,j}$ (see line 10 in Algorithm 1). The tracking table (named Cut in Algorithm 1) stores the value of the index k which minimizes $Tree[i, j]$ (see line 11 in Algorithm 1). It is the optimal decomposition value of $T_{i,j}$ into two subtrees. For a problem of size $n = 3$, the dependency graph partitioning and the DP table are illustrated in Figures 3a and 3b, respectively.

Knuth [4] noticed entries of tracking table Cut satisfy the *monotonicity property* such that $Cut[i, j - 1] \leq Cut[i, j] \leq Cut[i + 1, j]$ for all $0 \leq i < j \leq n$. This property means that computing $Cut[i, j]$ consists in finding all indices between $Cut[i, j - 1]$ and $Cut[i + 1, j]$, instead of between i and j as it is done in line 7 of Algorithm 1. He thus achieves to evaluate the $\Theta\left(n^2\right)$ subproblems in constant time. Algorithm 2 draws a big picture.

(a) Dependency graph   (b) DP table

Figure 3. Dependency graph and dynamic-programming table used to compute $Tree[0, 3]$

Algorithm 2 Knuth's sequential algorithm

1: **for** $i = 0$ to n **do**
2:     $Tree[i, i] \leftarrow q_i$;
3: **for** $d = 1$ to n **do**
4:     **for** $i = 0$ to $n - d + 1$ **do**
5:         $j \leftarrow n - d + 1$;
6:         $Tree[i, j] \leftarrow \infty$;
7:         **for** $k = Cut[i, j - 1]$ to $Cut[i + 1, j]$ **do**
8:             $c \leftarrow Tree[i, k] + Tree[k + 1, j] + w(i, j)$;
9:             **if** $c < Tree[i, j]$ **then**
10:                 $Tree[i, j] \leftarrow c$;
11:                 $Cut[i, j] \leftarrow k$;

One downside to the speedup of Knuth, however, is that the number of comparison operations for entries in the same diagonal varies from one entry to another compared to the classical version [4]. Therefore, when designing parallel algorithms based on Knuth's sequential algorithm, there is no guarantee that the processors will have the same load if entries on a diagonal are fairly distributed among them. It is the main parallelization constraint of this algorithm.

3. Dynamic graph model of the OBST problem

Kengne and Lacmou [16] showed that the OBST problem can be solved through the shortest path algorithms on weighted dependency graph. Indeed, they proposed a graph model called *dynamic graph* that is formulated from Equation (2). For a problem of size n, this graph is denoted by D_n and defined as follows :

Definition 3.1. Given a set of n sorted keys, a dynamic graph $D_n = (V, E \cup E')$ is

defined as a set of vertices,

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

a set of unit edges,

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

a set of jumps,

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

a weight function W such that

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

A square matrix of size n called *shortest path matrix*, and denoted by SP, is used to store in the cell $SP[i, j]$ the shortest path from node $(-1, -1)$ to (i, j). They showed that the computation of $Tree[i, j]$ is equivalent to search in D_n the shortest path from node $(-1, -1)$ to (i, j). Therefore, it is straightforward to prove that Knuth's sequential algorithm is equivalent to compute the shortest paths from $(-1, -1)$ to the other vertices in a dynamic graph D_n, incrementally, diagonal after diagonal, from left to right. Thus, the shortest path corresponds to the optimal binary search tree. Figure 4a shows a dynamic graph D_3 for a problem of size $n = 3$. It has the same form as the dependency graph between subproblems depicted in Figure 3a, with an additional node $(-1, -1)$.

Given a problem of size n and its corresponding dynamic graph D_n :

Theorem 3.2 (Duality Theorem). *If a shortest path from $(-1, -1)$ to (i, j) contains the jump $(i, k) \Rightarrow (i, j)$, then there is a dual shortest path containing the unit edge $(k + 1, j) \uparrow (i, j)$ when $0 \le i = k < j \le n$ and the jump $(k + 1, j) \Uparrow (i, j)$ when $0 \le i < k < j \le n$.*

Theorem 3.2 is fundamental because it makes it possible to avoid redundant computations when looking for the value of the shortest path of D_n's vertices. Indeed, for any vertex (i, j), among all its shortest paths containing jumps, only those that contain only horizontal jumps are evaluated. The input graph of our CGM-based parallel solution is therefore a subgraph of D_n denoted by D'_n, in which the set of edges from (i, k) to (i, j), such that $0 \le i = k < j \le n$, and from $(k + 1, j)$ to (i, j), such that $0 \le i < k < j \le n$, is removed. Figure 4b shows the dynamic graph D'_3.

(a) Dynamic graph D_3 (b) Dynamic graph D_3'

Figure 4. Dynamic graphs D_3 and D_3' for $n = 3$

4. Our CGM-based parallel solution using the four-splitting technique

4.1. Dynamic graph partitioning

The irregular partitioning technique consists in subdividing the shortest path matrix into submatrices (blocks) of varying size (irregular size) to enable a maximum of processors to remain active longer. The idea is to increase the number of blocks of diagonals whose this number is lower or equal to half of the first one through the block fragmentation technique. This technique aims to reduce the block size by dividing it into four subblocks. To minimize the number of communication rounds, it begins to subdivide the shortest path matrix with large-size blocks from the largest diagonal (the first diagonal of blocks) to the diagonal located just before the one whose number of blocks is half of the first one. Then, since the number of blocks per diagonal quickly becomes smaller than the number of processors, to increase the number of blocks of these diagonals and enable a maximum of processors to remain active, it fragments all the blocks belonging to the next diagonal until the last one to catch up or exceed by one notch the number of blocks of the first diagonal. It reduces the idle time of processors and promotes the load balancing. This process is repeated k times, after which the block sizes are no longer modify, and the rest of the partitioning becomes traditional because an excessive fragmentation would lead to a drastic rise in the number of communication rounds. After performing k fragmentations, a block belonging to lth level of fragmentation has been subdivided l times, $0 \leq l \leq k$.

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

By denoting $f(p) = \lceil \sqrt{2p} \rceil$, $\theta(n, p) = \lceil (n + 1)/f(p) \rceil$, and $\theta(n, p, l) = \lceil \theta(n, p)/2^l \rceil$, formally, we subdivide the shortest path matrix SP into blocks (denoted by $SM(i, j)$), and split the large-size blocks into four subblocks. Thus, a block $SM(i, j)$ belonging

9

Figure 5. Four-splitting technique of the shortest path matrix for $n = 31$, $k \in \{1, 2\}$, and $p \in \{3, 4, 5, 6, 7, 8\}$. For $p \in \{3, 4\}$, SP is partitioned into fifteen blocks and twenty-one subblocks when $k = 1$, and into twenty-four blocks and fifty-seven subblocks when $k = 2$. For $p \in \{5, 6, 7, 8\}$, SP is partitioned into nineteen blocks and thirty-six subblocks when $k = 1$, and into twenty-eight blocks and seventy-two subblocks when $k = 2$

to the lth level of fragmentation, such that $l < k$, is a $\theta(n, p, l) \times \theta(n, p, l)$ matrix and is subdivided into four subblocks of size $\theta(n, p, l+1) \times \theta(n, p, l+1)$. The blocks of the kth level of fragmentation are not splitting into four as these are the smallest blocks. Figures 5a, 5b, 5c, and 5d depict four scenarios of this partitioning for $n = 31$, $k \in \{1, 2\}$, and $p \in \{3, 4, 5, 6, 7, 8\}$. The number in each block represents the diagonal in which it belongs.

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

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

(2) all the blocks are not full when $n \bmod \left(2^k \times f(p)\right) \neq 0$, for example in Figure

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

Figure 6. Dependencies and extremities of a block $SM(i,j)$ after applying the four-splitting technique

    5b where $32 \bmod \left(2^2 \times 3\right) = 8 \neq 0$);

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

(4) one fragmentation increases up to $(\lceil f(p)/2 \rceil + 1)$ the number of diagonals;

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

4.2. Blocks' dependency analysis

Figures 6a and 6b show respectively an example of dependencies and extremities of a block $SM(i,j)$ after applying the four-splitting technique. The extremities of $SM(i,j)$ are defined by :

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

Figure 6a depicts eight points $(A,\ B,\ C,\ D,\ E,\ F,\ G,$ and $H)$ that identify blocks on which the block $SM(i,j)$ depends :

- $A = Tree\left[LUE_{i,j-3\times\theta(n,g,l)-1}\right]$
- $B = Tree\left[LUE_{i,j-\theta(n,g,l)-2}\right]$
- $C = Tree\left[LLE_{i,j-\theta(n,g,l)-2}\right]$

Figure 7. Snake-like mapping on six processors when $k = 1$

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

All lower blocks and subblocks (a part of a block) that are in the same row and column than the block $SM(i,j)$ are no longer absolutely required to evaluate $SM(i,j)$ as a consequence of the speedup of Knuth [4]. In Figure 6a for example, only the five most shaded blocks will be needed. In other cases, it could have been just four, three or two blocks. Moreover, it may happen that some nodes of blocks located in extremities are needed instead of the whole blocks. It depends on the input data. Therefore, the speedup of Knuth doesn't enable to estimate the exact load of a block before its evaluation [14]. However, the blocks on which the block $SM(i,j)$ depends aren't located on the same diagonal than $SM(i,j)$. Hence, they can be carried out in parallel.

4.3. Mapping blocks onto processors

The mapping used here consists in assigning the blocks of a given diagonal from the upper leftmost corner to the lower rightmost corner. First, the blocks of the first diagonal are assigned from processor P_0 to processor $P_{f(p)-1}$. Then, the process is repeated on the next diagonal starting with processor $P_{f(p)}$, and so on until a block has been assigned to each processor. The mapping starts again with processor P_0, and continues along the diagonals with a snake-like path until the last diagonal. Figure 7 illustrates this mapping on six processors.

The communication scheme corresponding to this mapping is simple and easy to implement. Moreover, this mapping enables processors to remain active as soon as possible and minimizes the number of communication rounds. It also ensures load

balancing because it enables some processors to evaluate at most one block more than others. For example, Figure 7 shows that only processor P_0 evaluates one more block than others. However, this mapping does not optimize communications insofar as a processor does not usually hold the upper blocks that are on the same row and column than the block that has been assigned to that processor.

4.4. Our CGM-based parallel algorithm

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

The four-splitting technique can be adapted to evaluate the subblocks that are part of a block in a non-progressive fashion compared to the k-block splitting technique proposed in [17]. Recall that when the latter is used, the number of subblocks (or k-blocks) drastically grows as the number of fragmentation rises. When a subblock is evaluated in a non-progressive fashion by a processor, it must receive all the data of subblocks that it needs to start computations. However, since the subblocks are numerous and small, computing and communicating them to processors that need them will lead to communication overhead as a huge amount of communication must be done to exchange data [17]. With the four-splitting technique, the blocks are subdivided into at most four, regardless of the number of fragmentation performed. This reduces the number of communication rounds and the number of steps to evaluate a whole block.

Denote the four subblocks of given block by LL, LU, RL, and RU, which respectively correspond to the subblock located in the leftmost lower corner, the leftmost upper corner, the rightmost lower corner, and the rightmost upper corner. Figures 8a, 8b, 8c, 8d, 8e, and 8f illustrate the different computation and communication steps of these subblocks by a processor :

(1) At step 0 in Figure 8a, no subblocks have started to be evaluated. This means that the processor is still receiving data of subblocks on which LL depends.
(2) At step 1 in Figure 8b, LL is computed but it is not communicated because a communication is not necessary at this step. Indeed, the processors which will receive LL will not be able to start the computation as soon as possible because the evaluation is done in a non-progressive fashion. The ideal would be to wait to compute LU before communicating them together.
(3) At step 2 in Figure 8c, the processor starts by receiving the lower subblocks that are in the same row than LU. Then, it computes LU. Finally, it communicates LL and LU to processors that need them. For example, the processor P_5 which evaluates the block of the second diagonal in Figure 7 will send LL and LU to P_1 and P_3.
(4) At step 3 in Figure 8d, the processor receives the lower subblocks that are in the same column than RL. Then, it computes RL. Finally, it communicates LL and RL to processors that need them. By taking the previous example, P_5 will send LL and RL to P_0 and P_2.
(5) At step 4 in Figure 8e, the processor computes RU. Thereafter, it communicates

Figure 8. Steps to evaluate the four subblocks of a given block when solving the OBST problem with the four-splitting technique. When a subblock is computed and needs to be communicated, it is colored in gray

---

Algorithm 3 Our CGM-based parallel algorithm based on the four-splitting technique

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

      RL and RU to processors that need them. By taking the previous example, P_5 will send RL and RU to P_1 and P_4.

(6) At step 5 in Figure 8f, all subblocks have been computed. The processor communicates LU and RU to processors that need them. By taking the previous example, P_5 will send LU and RU to P_3.

   Our CGM-based parallel algorithm based on the four-splitting technique to solve the OBST problem is given by Algorithm 3.

14

Theorem 4.1. *Our CGM-based parallel solution based on the four-splitting technique requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds in the worst case to solve the OBST problem.*

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

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

During the computation rounds where Knuth's sequential algorithm is used to evaluate blocks, evaluating each subblock of a block belonging to the lth level of fragmentation requires $\mathcal{O}\left(\frac{n^2}{2^{2(l+1)}\times(2p)^{2/2}}\right) = \mathcal{O}\left(\frac{n^2}{4^{l+1}\times p}\right)$ local computation time. So, the evaluation of each diagonal required :

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

The number of communication rounds is equal to :

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

Therefore, this algorithm requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds. $\square$

5. Experimental results

5.1. Experimental setups

This section highlights the results obtained from experimentations of our CGM-based parallel solutions based on the four-splitting technique to solve the OBST problem, and compares them with the best previous solutions [14,16]. These experimentations have been performed on the cluster dolphin of the MatriCS platform of the University of Picardie Jules Verne[1] and on our Raspberry Pi cluster (described hereunder). The MatriCS platform is composed of 48 nodes called thin nodes with 48×128GB of

---

RAM, and 12 named thick nodes with 12×512GB of RAM. Each node is made of two Intel Xeon Processor E5-2680 V4 (35M Cache, 2.40 GHz), and each of them consists of 14 cores. All nodes are interconnected with OmniPath links providing 100Gbps throughput. These algorithms have been implemented in the C programming language[2], and on the operating system CentOS Linux release 7.6.1810. The MPI library (OpenMPI version 1.10.4) has been used for inter-processor communication. These algorithms have been executed on five thin nodes. The results are presented following the different values of (n, p, k), where :

- n is the data size, with values in the set $\langle 4096, 8192, 12288, 16384, 20480, 24576, 28672, 32768, 36864, 40960 \rangle$.
- p is the number of processors, with values in the set $\langle 1, 32, 64, 96, 128 \rangle$. When $p = 1$, the sequential algorithm of Knuth [4] is carried out.
- k is the number of fragmentations performed, with values in the set $\langle 0, 1, 2, 3, 4, 5 \rangle$. When $k = 0$, our solution is similar to the one in [14].

Tables 1, 2, 3, 4, and 5 show the total execution time, the speedup, and the efficiency of our CGM-based parallel solutions to solve the OBST problem. In these tables, the solution based on the irregular partitioning technique is referred to *frag* and the one based on the four-splitting technique is referred to *4s*. Figures 9a, 9b, 10a, 10b, 11a, 11b, 12a, 12b, 13a, and 13b are drawn from the results obtained in Tables 1, 2, 3, 4, and 5.

5.2. *Evolution of the global communication time*

Figures 9a and 9b depict the global communication time of *frag* and *4s* while solving the OBST problem by performing one and two fragmentations. Recall that the global communication time is composed of the latency time of processors and the effective transfer time of data. These figures show that the global communication time of *frag* and *4s* is relatively lower than the one based on the regular partitioning (when $k = 0$). It gradually decreases as the number of fragmentations increases. For example, on thirty-two processors when $n = 40960$, the global communication time of *frag* (respectively *4s*) decreases on average by 3.87% (respectively 41.84%) when $k = 1$ and on average by 40.49% (respectively 54.83%) when $k = 2$. Indeed, when a fragmentation is performed, the number of blocks increase by dividing the current size of blocks into four to form the smaller-size blocks. It enables to minimize the latency time of processors and the effective transfer time of data since the smaller-size blocks take less time to evaluate and communicate compared to the larger-size blocks.

Figures 9a and 9b also show that the global communication time of *4s* is lower than *frag*. It decreases significantly when the number of fragmentations increases. These results were expected because *4s* minimizes the latency time of processors by enabling them to start evaluating subblocks as soon as the data they need are available. Grouping the subblocks into two before communicating also contributes to this minimization because it limits the number of messages to be exchanged in the network.

Figures 10a and 10b show that the global communication time and the overall computation time of *4s* gradually decrease while solving the OBST problem by successively performing five fragmentations. It can be noticed that until the fifth fragmentation the global communication time does not deteriorate. For example, on thirty-two processors when $n = 40960$, it decreases in average by 41.84% when $k = 1$, in average by

---

[2]The source codes are available here : `https://github.com/compiii/CGM-Sol-for-OBST`.

Figure 9. Global communication time for $n \in \{4096, \ldots, 40960\}$, $p \in \{32, \ldots, 128\}$, and $k \in \{0, 1, 2\}$ on the MatriCS platform

Figure 10. Comparison of the overall computation time and the global communication time for $n \in \{24576, 32768, 40960\}$, $p \in \{32, \ldots, 128\}$, and $k \in \{1, \ldots, 5\}$ while solving the OBST problem with the four-splitting technique on the MatriCS platform

54.83% when $k = 2$, in average by 58.18% when $k = 3$, in average by 59.43% when $k = 4$, and in average by 69.08% when $k = 5$. This is due to the sequential algorithm of Knuth [4] which is used to evaluate small subblocks that are close to the optimal solution. Since they do not require a high evaluation time, processors that need these subblocks will not have to wait long to receive them.

5.3. Evolution of the load-balancing of processors

Figures 11a and 11b compare the load imbalance on thirty-two processors by performing the difference between the average computation time of processors and the lowest (and highest) computation time among them. The processor with the lowest or highest computational load varies with the number of fragmentations performed. For example, when $k = 0$ and $k = 2$, respectively, P_7 and P_{30} have the lowest loads, and P_3 and P_{22} have the highest.

These figures show that the irregular partitioning technique of the dynamic graph balances the load of processors better than the regular partitioning technique of this graph due to the gradual reduction of the block sizes that enables processors to remain active as long as possible. Indeed, the fragmentation is performed on the blocks with the largest loads. So, some small-size blocks (which are in the upper diagonals) have higher loads than the large ones (which are in the lower diagonals). Thanks to the

17

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

Figure 11. Load imbalance of processors for $n \in \{24576, 32768, 40960\}$, $p = 32$, and $k \in \{0, 1, 2\}$ on the MatriCS platform

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

Figure 12. Total execution time for $n \in \{4096, \ldots, 40960\}$, $p \in \{32, \ldots, 128\}$, and $k \in \{0, 1, 2\}$ on the MatriCS platform

snake-like mapping which enables to distribute blocks onto processors in an equitable way, a processor can have in the worst case one more block than another. This greatly contributes to balancing the loads of processors. When $n = 40960$ and $k = 1$, the lowest load narrows down to 27.52% for *frag* and to 27.24% for *4s*; and the highest load increases up to 11.89% for *frag* and to 11.60% for *4s*. In the same vein when $k = 2$, the lowest load and the highest load decrease on average by 56.63% and 42.98% for *frag*, and on average by 55.73% and 42.77% for *4s*. Therefore, our CGM-based parallel solution based on the four-splitting technique (*4s*) kept the good performance of our CGM-based parallel solution based on the irregular partitioning technique (*frag*) with respect to the load-balancing of processors since they promote the load balancing when the number of fragmentations increases. These results were predictable since *4s* is based on the irregular partitioning technique and the snake-like mapping.

5.4. Evolution of the total execution time

Figures 10a, 10b, 12a, and 12b illustrate the total execution time of *frag* and *4s* while solving the OBST problem by performing one, two, three, four, and five fragmentations. These figures show that the minimization of the global communication time (due to the minimization of the latency time of processors) lead up to a reduction of the total execution time as the number of fragmentations rises. For example, on thirty-two

18

processors when $n = 40960$ and $k = 1$ (respectively $k = 2$) in Figure 10a, the total execution time decreases on average by 15.96% (respectively 47.87%) for *frag* and on average by 46.93% (respectively 59.37%) for *4s*. Similar observations can be made on one hundred and twenty-eight processors in Figure 10b; where when $n = 40960$ and $k = 1$ (respectively $k = 2$), the total execution time decreases on average by 35.40% (respectively 45.50%) for *frag* and on average by 42.18% (respectively 56.80%) for *4s*. It is then obvious to notice the huge impact that the global communication time has on the total execution time, and consequently on the speedup and the efficiency (as shown in Tables 1, 2, 3, and 4). The following results can be noticed :

- From $n = 4096$ to 40960 on thirty-two to one hundred and twenty-eight processors, the total execution time of *4s* is better than that of *frag* when performing one and two fragmentations. For example, on thirty-two processors when $n = 40960$ and $k = 1$ (respectively $k = 2$), the speedup and the efficiency are equal to 3.33 and 10.41% (respectively 5.37 and 16.78%) for *frag*, and increase up to 5.27 and 16.48% (respectively 6.89 and 21.53%) for *4s*.
- When performing three, four, and five additional fragmentations, the total execution time of *4s* keeps good performance, although from the fourth fragmentation the performance gain is not meaningful anymore. For example, on thirty-two processors when $n = 40960$, the speedup and the efficiency are equal to 7.54 and 23.55% when $k = 3$, and rise up to 7.74 and 24.17% when $k = 4$, up to 7.78 and 24.31% when $k = 5$.

From all this, we can deduce that it is better to use our CGM-based parallel solution based on the four-splitting technique to solve the OBST problem since it outperforms the one using the irregular partitioning technique and it is scalable as the data size, the number of processors, and the number of fragmentations rise.

5.5. What is the ideal number of fragmentations ?

This is a legitimate question because as shown earlier, on the MatriCS platform the total execution time of *4s* degrades from the fourth fragmentation while solving the MPP but keeps good performance until the fifth fragmentation while solving the OBST problem. One idea to determine the ideal number of fragmentations would be to run our CGM-based parallel solution based on the four-splitting technique to solve the OBST problem for example on another cluster. If we obtain the same result that was archived on the MatriCS platform then this could guide us to determine this number. Otherwise, we will deduce that the ideal number of fragmentations depends on the platform where the parallel algorithm is executed.

To achieve this goal, we have built a cluster consisting of four compute nodes. Each node is a Raspberry Pi 4 computer model B composed of a 64-bit quad-core Cortex-A72 processor with 8GB of RAM and an operating system Raspbian GNU/Linux 11. All nodes are interconnected with Cat8 Ethernet cable through a TP-Link router (model TL-SG108E) providing 16Gbps throughput and the inter-processor communication has been ensured by OpenMPI version 4.1.2.

Figures 13a and 13b compare the overall computation time and global communication time of *4s* on thirty-two processors while solving the OBST problem by performing one, two, three, four, and five fragmentations on the MatriCS platform and on our Raspberry Pi cluster. It is obvious to notice that whatever the cluster, *4s* progressively reduces the global communication time when the number of fragmen-

Figure 13. Comparison of the overall computation time and the global communication time for $n \in \{8192, 12288, 16384\}$, $p = 32$, and $k \in \{1, \ldots, 5\}$ while solving the OBST problem with the four-splitting technique on on the MatriCS platform and our Raspberry Pi cluster

tations rises. However on the Raspberry Pi cluster, the performance of *4s* degrades when adding the overall computation time at the fifth fragmentation for $n = 8192$ and $n = 12288$ because the global communication time did not reduce enough to keep the good performance. For example when $n = 12288$ in Table 5, the speedup is equal to 1.64 when $k = 1$, and increases up to 2.71 when $k = 2$, up to 3.03 when $k = 3$, up to 3.33 when $k = 4$, but the speedup narrows down to 3.24 when $k = 5$. This result is due to the fact that the Raspberry Pi cluster does not have a communication network as fast as the MatriCS platform. Indeed, an additional fragmentation is no longer necessary from the fourth fragmentation. Thus, since the network is not very fast, excessive communication caused a drop in performance. In summary, we deduce that the ideal number of fragmentations depends on the architecture where the CGM-based parallel solution is executed.

Another relevant observation can be seen on the Raspberry Pi cluster in Figure 13b, where the total execution time of *4s* gradually decreases after five successive fragmentations when $n = 16384$. As shown in Table 5, the speedup is equal to 2.20 when $k = 1$, and increases up to 2.64 when $k = 2$, up to 2.75 when $k = 3$, up to 2.95 when $k = 4$, and up to 3.08 when $k = 5$. This is due to the fact that when the input data size increases, fragmentations are more and more required until the performance gain is not significant or drops. However, as seen earlier, the performance of *4s* degrades at the fifth fragmentation for $n = 8192$ and $n = 12288$. Consequently, we infer that the ideal number of fragmentations also depends on the input data size.

6. Conclusion

We presented in this paper a CGM-based parallel solution to solve the optimal binary search tree problem using the four-splitting technique. It is a partitioning technique consisting of subdividing the dynamic graph into subgraphs (or blocks) of variable size and splitting large-size blocks into four subblocks. It enables processors to start the evaluation of blocks as soon as possible to minimize their latency time, which is the largest part of the global communication time. Our CGM-based parallel solution requires $\mathcal{O}\left(n^2/\sqrt{p}\right)$ execution time with $\mathcal{O}\left(k\sqrt{p}\right)$ communication rounds. Experimental results showed a good agreement with theoretical predictions since it significantly decreases the global communication time compared to the solution using the irregu-

lar partitioning technique [16] (and thus minimizes the total execution time). It also minimizes the number of communication rounds and balances the load of processors as well as the irregular partitioning technique. We also conducted experimentations to determine the ideal number of times the size of the blocks must be subdivided. The results showed that this number depends on the input data size and the architecture where the solution is executed. It would be interesting to undertake the challenge to determine this number before starting or during computations since it depends on the architecture where the solution is executed. It would be also interesting to apply our partitioning techniques on shared-memory architectures and GPU architectures.

Disclosure statement

The authors declare no conflicts of interest associated with this manuscript.

Data availability statement

The authors confirm that the data supporting the findings of this work are available within the manuscript.

References

[1] Wah BW, Li GJ. Systolic Processing for Dynamic Programming Problems. Circuits, Systems, and Signal Processing. 1988;7(2):119–149.

[2] Cormen TH, Leiserson CE, Rivest RL, et al. Introduction to Algorithms. 3rd ed. The MIT Press; 2009.

[3] El-Qawasmeh E. Word Prediction Using a Clustered Optimal Binary Search Tree. Information Processing Letters. 2004;92(5):257–265.

[4] Knuth DE. Optimum Binary Search Trees. Acta Informatica. 1971;1(1):14–25.

[5] Godbole SS. On Efficient Computation of Matrix Chain Products. IEEE Transactions on Computers. 1973;100(9):864–866.

[6] Karpinski M, Rytter W. On a Sublinear Time Parallel Construction of Optimal Binary Search Trees. In: the 1994 International Symposium on Mathematical Foundations of Computer Science; Košice, Slovakia; 1994. p. 453–461.

[7] Karpinski M, Larmore L, Rytter W. 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, Pennsylvania, USA; 1996. p. 36–41.

[8] Craus M. Parallel and Distributed Solutions for the Optimal Binary Search Tree Problem. In: the 2002 International Conference on Coding and Cryptology; Mangalia, Dobroudja, Romania; 2002. p. 103–117.

[9] Wani MA, Ahmad M. Statically Optimal Binary Search Tree Computation Using Non-Serial Polyadic Dynamic Programming on GPU's. International Journal of Grid and High Performance Computing. 2019;11(1):49–70.

[10] Bielecki W, Blaszynski P, Poliwoda M. 3D Parallel Tiled Code Implementing a Modified Knuth's Optimal Binary Search Tree Algorithm. Journal of Computational Science. 2021; 48(1).

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

[12] Dehne F, Ferreira A, Cáceres E, et al. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. Algorithmica. 2002;33(2):183–200.

[13] Kechid M, Myoupo JF. A Coarse Grain Multicomputer Algorithm Solving the Optimal Binary Search Tree Problem. In: the Fifth International Conference on Information Technology: New Generations; Las Vegas, Nevada, USA; 2008. p. 1186–1189.

[14] Myoupo JF, Kengne TV. Parallel Dynamic Programming for Solving the Optimal Search Binary Tree Problem on CGM. International Journal of High Performance Computing and Networking. 2014;7(4):269–280.

[15] Kengne TV, Myoupo JF, Dequen G. High Performance CGM-based Parallel Algorithms for the Optimal Binary Search Tree Problem. International Journal Grid High Performance Computing. 2016;8(4):55–77.

[16] Kengne TV, Lacmou ZJ. An Efficient CGM-Based Parallel Algorithm for Solving the Optimal Binary Search Tree Problem Through One-to-All Shortest Paths in a Dynamic Graph. Data Science and Engineering. 2019;4(2):141–156.

[17] Lacmou ZJ, Kengne TV, Myoupo JF. High-Performance CGM-Based Parallel Algorithms for Minimum Cost Parenthesizing Problem. The Journal of Supercomputing. 2021;.

[18] Nagaraj S. Optimal Binary Search Trees. Theoretical Computer Science. 1997;188(1-2):1–44.

[19] Knuth DE. The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed. Addison-Wesley; 1998.

Table 1. Total execution time (in seconds) for $n \in \{4096,\ldots,40960\}$, $p = 32$, and $k \in \{0,\ldots,5\}$ on the MatriCS platform

n	$p = 1$	frag			4s				
		$k = 0$	$k = 1$	$k = 2$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
4096	62.34	20.58	16.68	13.11	11.50	9.90	8.94	8.58	8.52
8192	514.32	180.82	136.43	100.02	97.34	77.30	70.76	69.10	68.64
12288	1784.19	621.52	476.46	343.27	338.27	266.48	243.43	236.72	235.48
16384	4315.43	1582.57	1142.21	818.11	813.47	634.41	585.65	572.21	568.37
20480	8546.86	2919.91	2308.19	1600.75	1605.26	1244.97	1147.37	1123.66	1116.53
24576	14899.60	5970.20	3920.49	2785.83	2793.95	2151.68	1977.78	1937.93	1930.08
28672	23802.78	8527.82	6078.21	4428.91	4458.84	3444.42	3155.25	3089.17	3076.31
32768	33263.71	12533.26	10130.78	6607.36	6682.07	5153.27	4718.72	4616.36	4587.60
36864	48356.46	17658.48	13514.64	9451.97	9578.16	7353.35	6729.28	6572.38	6529.41
40960	69722.12	24910.11	20933.65	12986.77	13219.62	10120.25	9252.37	9013.49	8961.54

Table 2. Speedup and efficiency (in %) for $n \in \{4096,\ldots,40960\}$, $p = 32$, and $k \in \{1,\ldots,5\}$ on the MatriCS platform

$n \backslash k$	Speedup							Efficiency						
	frag		4s					frag		4s				
	1	2	1	2	3	4	5	1	2	1	2	3	4	5
4096	3.74	4.75	5.42	6.29	6.97	7.27	7.32	11.68	14.85	16.94	19.67	21.79	22.71	22.87
8192	3.77	5.14	5.28	6.65	7.27	7.44	7.49	11.78	16.07	16.51	20.79	22.71	23.26	23.42
12288	3.74	5.20	5.27	6.70	7.33	7.54	7.58	11.70	16.24	16.48	20.92	22.90	23.55	23.68
16384	3.78	5.27	5.30	6.80	7.37	7.54	7.59	11.81	16.48	16.58	21.26	23.03	23.57	23.73
20480	3.70	5.34	5.32	6.87	7.45	7.61	7.65	11.57	16.69	16.64	21.45	23.28	23.77	23.92
24576	3.80	5.35	5.33	6.92	7.53	7.69	7.72	11.88	16.71	16.67	21.64	23.54	24.03	24.12
28672	3.92	5.37	5.34	6.91	7.54	7.71	7.74	12.24	16.80	16.68	21.60	23.57	24.08	24.18
32768	3.28	5.03	4.98	6.45	7.05	7.21	7.25	10.26	15.73	15.56	20.17	22.03	22.52	22.66
36864	3.58	5.12	5.05	6.58	7.19	7.36	7.41	11.18	15.99	15.78	20.55	22.46	22.99	23.14
40960	3.33	5.37	5.27	6.89	7.54	7.74	7.78	10.41	16.78	16.48	21.53	23.55	24.17	24.31

Table 3. Total execution time (in seconds) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{0,\ldots,5\}$ on the MatriCS platform

	frag		4s					
p\k	1	2	0	1	2	3	4	5
				$n = 36864$				
64	7610.84	6586.99	11671.40	6849.69	5268.74	4744.92	4609.23	4492.70
96	6573.25	5552.72	10128.02	5811.06	4336.53	3960.81	3863.48	3846.97
128	5788.29	5126.66	8951.40	5166.80	3858.00	3488.67	3407.10	3391.69
				$n = 40960$				
64	10456.06	9057.29	16045.47	9564.80	7256.84	6333.78	6208.79	6177.57
96	9031.15	7813.92	13928.60	7980.85	5954.35	5415.29	5299.00	5276.19
128	7949.95	6707.94	12305.95	7115.22	5316.16	4815.67	4688.91	4670.79

Table 4. Speedup and efficiency (in %) for $n \in \{36864, 40960\}$, $p \in \{64, 96, 128\}$, and $k \in \{1,\ldots,5\}$ on the MatriCS platform

Speedup

	frag		4s				
p\k	1	2	1	2	3	4	5
			$n = 36864$				
64	6.35	7.34	7.06	9.18	10.19	10.49	10.76
96	7.36	8.71	8.32	11.15	12.21	12.52	12.57
128	8.35	9.43	9.36	12.53	13.86	14.19	14.26
			$n = 40960$				
64	6.67	7.70	7.29	9.61	11.01	11.23	11.29
96	7.72	8.92	8.74	11.71	12.88	13.16	13.21
128	8.77	10.39	9.80	13.12	14.48	14.87	14.93

Efficiency

	frag		4s				
p\k	1	2	1	2	3	4	5
			$n = 36864$				
64	9.93	11.47	11.03	14.34	15.92	16.39	16.82
96	7.66	9.07	8.67	11.62	12.72	13.04	13.09
128	6.53	7.37	7.31	9.79	10.83	11.09	11.14
			$n = 40960$				
64	10.42	12.03	11.39	15.01	17.20	17.55	17.63
96	8.04	9.29	9.10	12.20	13.41	13.71	13.77
128	6.85	8.12	7.66	10.25	11.31	11.62	11.66

Table 5. Total execution time (in seconds) and speedup for $n \in \{4096,\ldots,16384\}$, $p \in \{1, 32\}$, and $k \in \{1,\ldots,5\}$ while solving the OBST problem with the four-splitting technique on our Raspberry Pi cluster

	Total execution time						Speedup				
n	p=1	k=1	k=2	k=3	k=4	k=5	k=1	k=2	k=3	k=4	k=5
4096	152.45	70.53	61.74	50.45	51.68	52.24	2.16	2.47	3.02	2.95	2.92
8192	1252.78	553.87	425.86	415.23	385.80	391.02	2.26	2.94	3.02	3.25	3.20
12288	4384.93	2670.32	1615.90	1448.08	1318.49	1351.73	1.64	2.71	3.03	3.33	3.24
16384	10660.09	4856.32	4034.42	3870.51	3610.49	3464.09	2.20	2.64	2.75	2.95	3.08