PARALLEL RANGE MINIMA ON COARSE GRAINED MULTICOMPUTERS*

H. MONGELLI
Departamento de Computação e Estatística
Universidade Federal do Mato Grosso do Sul
Caixa Postal 549
Campo Grande, MS, 79.070-900 Brazil

and

S. W. SONG
Departamento de Ciência da Computação
Universidade de São Paulo
Rua do Matão, 1010
São Paulo, SP, 05508-900, Brazil

ABSTRACT

Given an array of n real numbers $A = (a_0, a_1, \ldots, a_{n-1})$, define $MIN(i,j) = \min\{a_i, \ldots, a_j\}$. The range minima problem consists of preprocessing array A such that queries $MIN(i,j)$, for any $0 \leq i \leq j \leq n-1$ can be answered in constant time. Range minima is a basic problem that appears in many other important graph problems such as lowest common ancestor, Euler tour, etc. In this work we present a parallel algorithm under the CGM model (coarse grained multicomputer), that solves the range minima problem in $O(n/p)$ time and constant number of communication rounds. The communication overhead involves the transmission of p numbers (independent of n). We show promising experimental results with speedup curves approximating the optimal for large n.

Keywords: Range minima, coarse grainded multicomputer, parallel algorithms, lowest common ancestor.

1. Introduction

Given an array of n real numbers $A = (a_0, a_1, \ldots, a_{n-1})$, define $MIN(i,j) = \min\{a_i, \ldots, a_j\}$. The range minima problem consists of preprocessing array A such that queries $MIN(i,j)$, for any $0 \leq i \leq j \leq n-1$ can be answered in constant time.

The parallel computing model used in this work is the CGM (coarse grained multicomputer). The CGM(n,p) model consists of an input of size $O(n)$ and p processors with point-to-point communication. An algorithm under this model consists of supersteps that involve a local computing round and a communication round between processors. In a communication round, each processor can send and receive $O(n/p)$ data.

This is a more realistic model than the PRAM model for most real parallel computers. The number of communication rounds is an important issue in the design of algorithms. It has been introduced in [7, 8, 9], where algorithms using a constant

*Supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) Proc. No. 98/06138-2, CNPq Proc. No. 52 3778/96-1, CNPq/NSF Collaborative Research Program Proc. No. 680037/99-3, and CAPES/PICDT.

number of communication rounds were presented for several computational geometry problems. Algorithms requiring $O(\log p)$ communication rounds have been presented for the solution of several graph problems [6].

In this work we consider the range minima problem. We present an algorithm under the CGM(n,p) model that solves the problem requiring a constant number of communication rounds and $O(n/p)$ computation time. In the PRAM model, there is an optimal algorithm of $O(\log n)$ time [13]. This algorithm, however, is not immediately transformable to the CGM model. Berkman et al. [4] describe an optimal algorithm of $O(\log\log n)$ time for a PRAM CRCW. This algorithm motivates the CGM algorithm presented in this paper. We do not, however, compare the performance of the proposed algorithm with that of Berkman et al. [4] experimentally. This is because Berkman's PRAM algorithm uses a large number of processors $(n/(\log\log n))$ as opposed to the proposed algorithm, which is more realistic in terms of the number of processors.

Range minima is a basic problem and appears in many graph problems. In [13], range minima is used together with Euler tour in the design of a PRAM algorithm for the problem of lowest common ancestor (LCA). LCA, in turn, is used in such graph algorithms as open-ear decomposition, biconnectivity, and planarity problems [17]. In addition to the LCA problem, range minima is also used in the pattern matching with scaling problem [3, 11, 16].

In Section 2, we present the CGM model and define prefix minimum, suffix minimum, and the LCA problem. In Section 3, we describe the sequential algorithms of Gabow et al. [11] and Alon and Schieber [2] for the range minima problem, to be used in the local computation round in each processor in the CGM algorithm. In Section 4, we present the CGM algorithm for the range minima problem, together with the proof of its time complexity and number of communication rounds. In Section 5, we describe how we answer queries $MIN(i,j)$ with the data structure obtained by the CGM algorithm. Section 6 presents promising experimental results with good speedup for large n. Conclusions are given in Section 7. This is the extended version of [15] with new, complete experimental results, including time and speedup curves.

2. Preliminaries

In this section we describe the parallel computing model and some definitions used in the design of the proposed algorithm.

2.1. CGM (Coarse Grained Multicomputer) Model

The CGM model was proposed by Dehne [7, 8, 9]. It is similar to the BSP (bulk-synchronous parallel) model [19]. However, it uses only two parameters: the input size n and the number of processors p. Each processor has a local memory of size $O(n/p)$. The term *coarse grained* means the size of the local memory is much larger than $O(1)$. Dehne et al. [9] define "much larger" as $n/p \geq p$. Each processor is connected by a router that can send messages in a point-to-point fashion. A

CGM algorithm consists of an alternate sequence of computing and communication rounds separated by a barrier synchronization.

A computing round is equivalent to a superstep in the BSP model. In a computing round, we usually use the best sequential algorithm in each processor to process its data locally. In a communication round, each processor exchanges a maximum of $O(n/p)$ data with other processors. In the CGM model, the communication cost is modeled by the number of communication rounds. Some algorithms for computational geometry and graph problems require a constant number or $O(\log p)$ communication rounds [6, 7, 8, 9, 10]. Contrary to a PRAM algorithm, which is often designed for $p = O(n^k)$, with $k \in \mathcal{N}$, and with each processor receiving a small number of input data, in this model we consider the more realistic cases where $n \gg p$. The CGM model is particularly suitable in current parallel machines in which the global computing speed is considerably larger than the global communication speed.

2.2. Definitions

2.2.1. Prefix Minimum and Suffix Minimum

These are particular cases of prefix sums and suffix sums, where we consider the minimum operator [17].

Definition 1 *Consider an array $A = (a_1, a_2, \ldots, a_n)$ of n real numbers. The prefix minimum array is the array $P = (\min\{a_0\}, \min\{a_0, a_1\}, \ldots, \min\{a_0, a_1, \ldots, a_{n-1}\})$. The suffix minimum array is the array $S = (\min\{a_0, a_1, \ldots, a_{n-1}\}, \min\{a_1, \ldots, a_{n-1}\}, \ldots, \min\{a_{n-2}, a_{n-1}\}, \min\{a_{n-1}\})$.*

2.2.2. The LCA Problem

Definition 2 *The lowest common ancestor of two vertices u and v of a rooted tree is the vertex w that is a common ancestor of u and v and that is farthest from the root. We denote $w = LCA(u, v)$.*

Definition 3 *The LCA problem consists of preprocessing a rooted tree such that queries of the type $LCA(u, v)$, for any vertices u and v of the tree, can be answered in constant time.*

3. Sequential Algorithms

The algorithm for the range minima problem in the CGM model uses two sequential algorithms that are executed using local data in each processor. These algorithms were presented by Gabow et al. [11] and Alon and Schieber [2], described in the following.

3.1. Sequential Algorithm of Gabow et al.

This sequential algorithm uses the Cartesian tree data structure, introduced in [20]. It also uses an algorithm for the lowest common ancestor problem.

Definition 4 (Cartesian tree [20]) *Given an array $A_{0,n-1} = (a_0, a_1, \ldots, a_{n-1})$ of n distinct real numbers, the Cartesian tree of $A_{0,n-1}$ is a binary tree with nodes labeled by the elements of Array A. The root of the tree has as label $a_m = \min\{a_0, a_1, \ldots, a_{n-1}\}$. Its left subtree is a Cartesian tree of $A_{0,m-1} = (a_0, a_1, \ldots, a_{m-1})$, and its right subtree is a Cartesian tree of $A_{m+1,n-1} = (a_{m+1}, \ldots, a_{n-1})$. The Cartesian tree of an empty array is the empty tree.*

In Fig. 1 we present an example of a Cartesian tree of the array $(10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9)$.

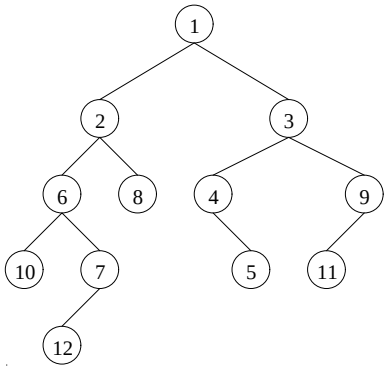

Fig. 1: Cartesian tree corresponding to the array $(10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9)$.

In this algorithm we need also an algorithm for the LCA problem. The algorithm for the range minima problem can be given as follows.

ALGORITHM 1: Range Minima (Gabow et al.)
Input: array $A = (a_0, a_1, \ldots, a_{n-1})$ of n real numbers.
Output: a data structure that answers queries $MIN(i, j)$ in constant time.

1. Build the Cartesian tree of A.
2. Use a linear sequential algorithm for the LCA problem using the Cartesian tree.

The construction of the Cartesian tree takes linear time [11]. Linear sequential algorithms for the LCA problem can be found in [5, 12, 18]. Any query $MIN(i, j)$ is answered as follows.

Processing queries. From the recursive definition of the Cartesian tree, the value of $MIN(i, j)$ is the value of the LCA of a_i and a_j. Thus, each range minima query can be answered in constant time through a query of LCA in the Cartesian tree.

Therefore, the range minima problem is solved in linear time.

3.2. Sequential Algorithm of Alon and Schieber

The algorithm of Alon and Schieber [2] has $O(n \log n)$ time complexity. In spite of its non-linear complexity, this algorithm is crucial in the description of the CGM algorithm, as will be seen in Section 4. Without loss of generality, we consider n to be a power of 2.

ALGORITHM 2: Range Minima (Alon and Schieber)
Input: array $A = (a_0, a_1, \ldots, a_{n-1})$ of n real numbers.
Output: a data structure that answers queries of the type $MIN(i, j)$ in constant time.

1. Construct a complete binary tree T with n leaves.
2. Associate the elements of A to the leaves of T.
3. For each node v of T, compute the arrays P_v and S_v, the prefix minimum, and the suffix minimum arrays, respectively, of the leaves of the subtree with root v.

Tree T constructed by Algorithm 2 will be called *PS-tree*. Fig. 2 illustrates the PS-tree generated by this algorithm.

Let us introduce another notation.

Notation 1 *Consider a complete binary tree and a given tree node v. Denote by $a_r, a_{r+1}, \ldots, a_s$ the leaves below v that are descendants of v. For a given j such as $r \leq j \leq s$, define*

$$Pos(a_j) = j - r.$$

Processing the queries. Queries of type $MIN(i, j)$, $0 \leq i \leq j \leq n-1$, are answered as follows. We find $w = \text{LCA}(a_i, a_j)$ in T. Let v and u be the left and right sons of w, respectively. Then, $MIN(i, j)$ is the minimum between the value of $S_v[\text{Pos}(a_i)]$ and the value of $P_u[\text{Pos}(a_j)]$.

PS-tree is a complete binary tree. In [13, 17] it is shown that a LCA query in a complete binary tree can be answered in constant time.

4. The CGM Algorithm for the Range Minima Problem

In the design of the CGM algorithm, we want to reduce the number of communication rounds and concentrate on the computation of local data in the processors.

Based on the sequential algorithms seen in the previous section, we can design a CGM algorithm for the range minima problem. This algorithm is executed in $O(n/p)$ time and uses $O(1)$ communication rounds. The main difficulty is how to store the required data structure among the processors so that the queries can be done in constant time without violating the limit of $O(n/p)$ memory required by the CGM model.

The CGM(n, p) model requires a minimum of $O(n/p)$ memory in each processor. Unfortunately, there is nothing that can be done if the amount of available memory is less than this minimum. First, we need sufficient memory to store the input data.

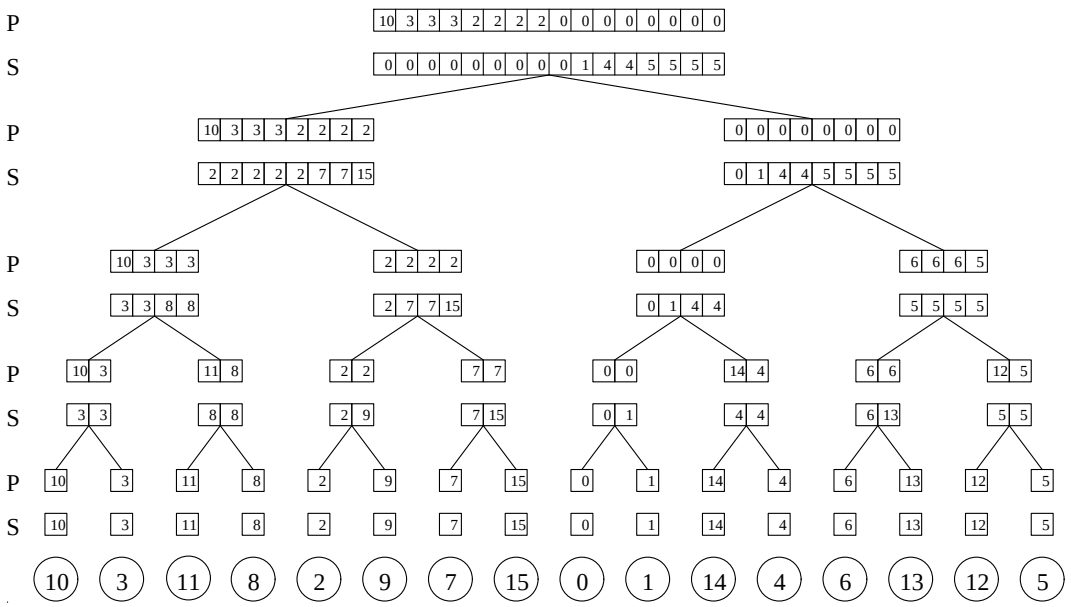

Fig. 2: PS-tree generated by Algorithm 2 for a particular array.

We also need memory to construct the proposed data structures in each processor in order to provide constant time response time in range minima queries.

In the description of the algorithm, we consider the following notation.

Notation 2 *Given an array* $A = (a_0, a_1, \ldots, a_{n-1})$*, we write* $A[i] = a_i$*,* $1 \leq i \leq n$*, and* $A[i \ldots j] =$ *subarray* $(a_i, \ldots, a_j)$*,* $1 \leq i \leq j \leq n$*.*

The idea of the algorithm is based on how queries $MIN(i, j)$ can be answered. Consider p processors denoted by $0, 1, \ldots, p-1$. Without loss of generality, we assume p to be a power of 2. Each processor stores n/p contiguous positions of the input array. Thus, given $A = (a_0, a_1, \ldots, a_{n-1})$, processor i stores subarray $A_i = (a_{i(n/p)}, \ldots, a_{(i+1)(n/p)-1})$, for $0 \leq i \leq p-1$. MIN(i, j) is answered depending on the location of a_i and a_j in the processors. We have the following cases.

1. If a_i and a_j are in the same processor, the problem domain is reduced to the subarray stored in that processor. Thus, we need a data structure to answer this type of query in constant time. This data structure is obtained in each processor by Algorithm 1.

2. If a_i and a_j are in distinct processors $\bar{i}$ and $\bar{j}$ (without loss of generality, $\bar{i} < \bar{j}$), respectively, we have two subcases:

   (a) If $\bar{i} = \bar{j} - 1$, that is a_i and a_j are in neighbor processors, $MIN(i, j)$ then corresponds to the minimum between the minimum of a_i through the end of array $A_{\bar{i}}$ and the minimum of the beginning of array $A_{\bar{j}}$ to a_j. These minima can again be determined by the data structure obtained by Algorithm 1. To determine the minimum of the minima, we need one

communication round.

(b) If $\bar{i} < \bar{j}-1$, $MIN(i,j)$ corresponds to the minimum among the minimum of subarray $A_{\bar{i}}[i-\bar{i}(n/p)-1 \ldots (\bar{i}+1)(n/p)-1]$, the minimum of subarray $A_{\bar{j}}[\bar{j}(n/p) \ldots j-\bar{j}(n/p)-1]$ and the minima of subarrays $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$. The first two minima are obtained as in the previous subcase. The minima of subarrays $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$ are easily obtained by using the Cartesian tree of the array of minima of all subarrays $A_k, 0 \leq k \leq p-1$. This amounts to the range minima problem restricted to an array of p numbers. Thus, we need a data structure to answer these queries in constant time. As the array of minima contains only p values, this data structure can be obtained by Algorithm 2.

The difficulty of case 2(b) is that we cannot construct the PS-tree explicitly in each processor as described in Section 3.2, since this would require a memory of size $O(p \log p)$, larger than the memory requirement in the CGM model, which is $O(n/p)$, with $n/p \geq p$. To overcome this difficulty we store only partial information of the PS-tree T in each processor. We construct arrays P' and S' of $\log p + 1$ positions in each processor, described as follows.

Let us describe this construction in processor i, with $0 \leq i \leq p-1$. Let b_i be the value of the minimum of array A_i. Let v be any node of T such that the subtree with root v has b_i as leaf, and let d_v be the *depth of* v in T, that is the path length from the root to v, as defined in [1]; and l_v be the *level of* v, which is the height of the tree minus the depth of v, also as defined in [1] ($l_v = \log p - d_v$, because tree T has height $\log p$). Array P' (respectively, S') contains in position l_v the value of array P_v (respectively, S_v), of the level l_v of T, in the position corresponding to the leaf b_i. In other words, we have $P'[l_v] = P_v[i \bmod 2^{l_v}]$.

Fig. 3 illustrates the correspondence between arrays P' and S' stored in each processor and the PS-tree constructed by Algorithm 2. In Fig. 3(b), the boldface positions in arrays S in each level of the tree correspond to the suffix minimum of $b_0 = 3$ in each level. In this way, we obtain array S' in P_0 (Fig. 3(c)).

In the following, we give a high-level description of the algorithm.

ALGORITHM 3: Range Minima (CGM model)
Input: array $A = (a_0, a_1, \ldots, a_{n-1})$ with n real numbers.
Output: a data structure to answer queries $MIN(i,j)$ in constant time.
{Each processor receives n/p contiguous positions of Array A, partitioned into subarrays $A_0, A_1, \ldots, A_{p-1}$.}

1. Each processor i executes sequentially Algorithm 1.

2. Comment: Each processor constructs an array $B = (b_i)$ of size p, containing the minima of the subarrays stored in each processor.

   2.1. Each processor i calculates $b_i = \min A_i = \min\{a_{i(n/p)}, \ldots, a_{(i+1)(n/p)-1}\}$.

   2.2. Each processor i sends b_i to the other processors.

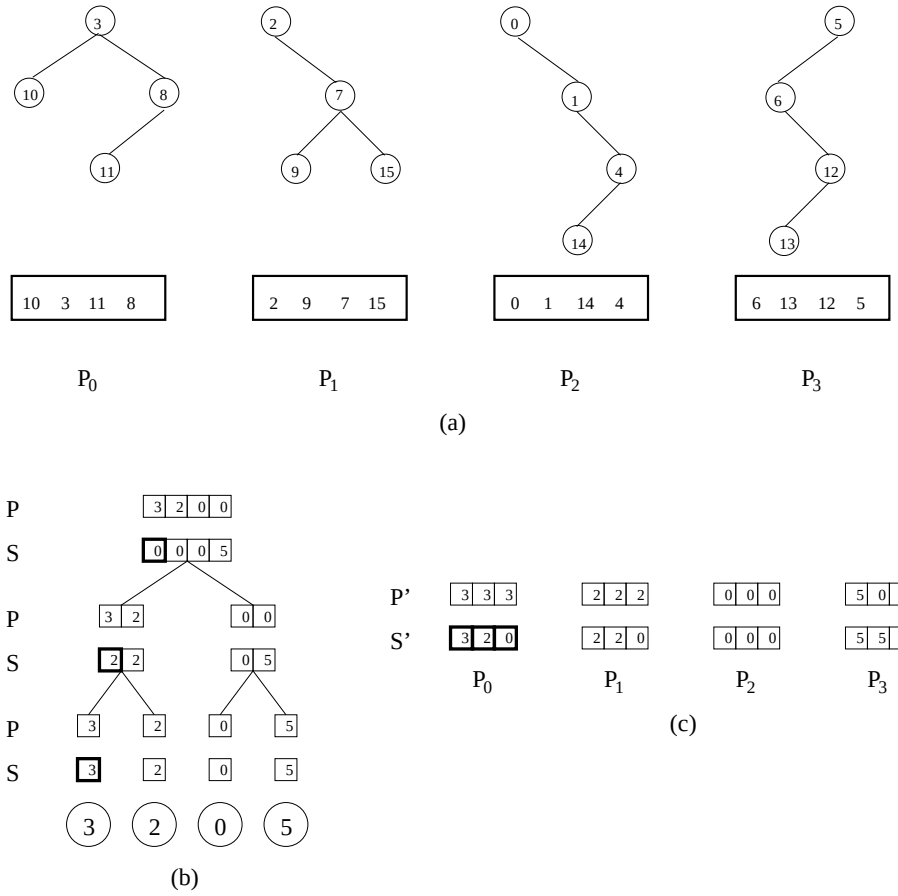

Fig. 3: Execution of Algorithm 3 using the array (10, 3, 11, 8, 2, 9, 7, 15, 0, 1, 14, 4, 6, 13, 12, 5). (a) The data distributed in the processors and the corresponding Cartesian trees. (b) PS-trees constructed by Algorithm 2 for the array $(3, 2, 0, 5)$ of the minima of processors. (c) Arrays P' and S' constructed by Step 3 of Algorithm 3 corresponding to arrays P and S of T.

2.3. Each processor i puts in b_k, the value received from processor k, $k \in \{0, \ldots, p-1\} \setminus \{i\}$.

3. Each processor i executes procedures Construct_P' and Construct_S' (see below) .

Given array B, the following procedure constructs array P' in processor i, for $0 \leq i \leq p-1$. This procedure constructs P' in $O(p)$ $(= O(n/p))$ time using only local data. The construction of array S' is done in a symmetric way, considering array B in reverse order.

PROCEDURE 4: Construct_P'.
Input: array $B = (b_0, b_1, \ldots, b_{p-1})$ with p real numbers.

Output: array P' of $\log p + 1$ positions.

1. $P'[0] \leftarrow b_i$
2. $pointer \leftarrow i$
3. $inorder \leftarrow 2 * i + 1$
4. **for** $k \leftarrow 1$ **until** $\log p$ **do**
5. $\quad P'[k] \leftarrow P'[k-1]$
6. $\quad$ **if** $\lfloor inorder/2^k \rfloor$ is odd
7. $\quad\quad$ **then for** $l \leftarrow 1$ **until** 2^{k-1} **do**
8. $\quad\quad\quad\quad pointer \leftarrow pointer - 1$
9. $\quad\quad\quad\quad$ **if** $P'[k] > B[pointer]$
10. $\quad\quad\quad\quad\quad$ **then** $P'[k] \leftarrow B[pointer]$

To simplify the correctness proof of this procedure, we consider p to be a power of 2 and that, in each processor i, array B contains leaves of a complete binary tree, as in the description of Algorithm 2. We do not store an entire array in each internal node of this tree, but only the $\log p + 1$ positions of the prefix minimum array in each level corresponding to position i of array B.

The value of the variable *inorder* of Line 3 is the value of the in-order number of the leaf containing b_i, obtained in an in-order traversal of the tree nodes. It can easily be seen that these values, from the left to the right, are the odd numbers in the interval $[1, 2p-1]$.

Theorem 1 *Procedure 4 correctly computes array P', for each processor i, $0 \leq i \leq p-1$.*

Proof. For a processor i, $0 \leq i \leq p-1$, we prove the following invariant at each iteration of the for loop of Line 4:

At each iteration k, let T_k be the subtree that contains b_i and has root at level k. $P'[k]$ stores position i of the minimum prefix array of the subarray of B corresponding to the leaves of subtree T_k and the variable *pointer* contains the index, in B, of the leftmost leaf of T_k.

Initially, we prove that the invariant holds before the first iteration. In this case, $k = 0$, $P'[k] = b_i$ and $pointer = i$ are valid, since subtree T_k is formed only by the leaf that contains b_i and that is at position i of B.

Now let us assume the variant to be valid immediately before the beginning of an iteration k and show that it remains valid at the end of this iteration.

In iteration k, let C_k be the subarray of B whose elements are the leaves of subtree T_k with root v. T_k contains b_i and v is at level k.

We analyze the two cases:

1. $\lfloor inorder/2^k \rfloor$ is even.

   Since we are considering a complete binary tree, the even value of $\lfloor inorder/2^k \rfloor$ means that the leaf corresponding to b_i is in the left subtree of v.

   In this case, the elements at the left of b_i in C_k are the same elements at the left of b_i in C_{k-1}. Thus, $P'[k] = P'[k-1]$. Furthermore, the leftmost leaf of T_k is also the leftmost leaf of tree T_{k-1} that contains b_i. Thus, the value of variable *pointer* does not change.

In Procedure 4, Line 5 performs the update $P'[k] \leftarrow P'[k-1]$, and this value of $P'[k]$ is not changed until the end of the iteration. The value of variable *pointer* also does not change.

Therefore, in this case, the invariant remains valid at the end of iteration k.

2. $\lfloor inorder/2^k \rfloor$ is odd.

   In this case, the leaf corresponding to b_i is in the right subtree of root v.

   The value stored at $P'[k-1]$ corresponds to the minimum suffix of subarray C_{k-1} of array B. The variable *pointer* stores the index of array B corresponding to the leftmost leaf of the subtree whose leaves are the elements of C_{k-1}. That is, this variable points to the first position of C_{k-1}.

   Subarray C_k, to be considered at iteration k, corresponds to the leaves of subtree T_k. Thus, this subarray is formed by the leaves of the left and right subtrees of v. (The leaves of the right subtree correspond to the elements of array C_{k-1}.)

   As this new subarray C_k presents values at the left of b_i that were not involved in the determination of $P'[k-1]$, the value of $P'[k]$ will be the minimum between $P'[k-1]$ and the values corresponding to the leaves of the left subtree of v.

   The for loop of Line 7 examines these positions, comparing the values with $P'[k]$ (initially equal to $P'[k-1]$) and updating it when needed. This is done by using variable *pointer* to consider these positions and at the end points to the first position of array C_k.

   Thus, at the end of this iteration, $P'[k]$ will contain the correct value of position i of the minimum suffix array B, and variable *pointer* will contain the index, in B, of the first element of C_k that corresponds to the leftmost leaf of T_k.

At the last iteration, $C_k = B$ and $P'[\log p]$ contains the value of position i of the minimum suffix array of B, and variable *pointer* contains the index of the first element of B.

Therefore, Procedure 4 correctly computes array P'. □

To compute S', we have a similar theorem with a similar proof.

Lemma 1 *Execution of Procedure 4 in each processor takes $O(n/p)$ sequential time.*

Proof The maximum number of iterations is $\log p$. In each iteration, the value of *pointer* is decremented or remains the same. The worst case is when $i = p - 1$. In this case, at each iteration the value of the variable *pointer* is updated. As the number of elements of B is p, the algorithm updates the value of *pointer* at most p times. Therefore, Procedure 4 is executed in $O(p) = O(n/p)$ sequential time. □

The following theorem summarizes the results obtained so far.

Theorem 2 *Algorithm 3 solves the range minima problem in $O(n/p)$ time using $O(1)$ communication rounds.*

Proof. Step 1 is executed in $O(n/p)$ sequential time and does not need communication. Step 2 runs in $O(n/p)$ sequential time and needs one communication round. By Theorem 1, Step 3 is executed in $O(n/p)$ sequential time and does not require communication. Therefore, Algorithm 3 solves the range minima problem in $O(n/p)$ time using $O(1)$ communication rounds. □

5. Query Processing

In this section, we show how to use the output of Algorithm 3 to answer queries $MIN(i, j)$ in constant time. A query $MIN(i, j)$ is handled as follows. Assume that i and j are known by all the processors and the result will be given by processor 0. If a_i and a_j are in the same processor, then $MIN(i, j)$ can be determined by Step 1 of Algorithm 3. Otherwise, suppose that a_i and a_j are in distinct processors $\bar{i}$ and $\bar{j}$, respectively, with $\bar{i} < \bar{j}$. Let $right(\bar{i})$ be the index in A of the rightmost element in array $A_{\bar{i}}$, and $left(\bar{j})$ the index in A of the leftmost element in array $A_{\bar{j}}$. Calculate $MIN(i, right(\bar{i}))$ and $MIN(left(\bar{j}), j)$, using Step 1. We have two cases:

1. If $\bar{j} = \bar{i} + 1$ then $MIN(i, j) = \min\{MIN(i, right(\bar{i})), MIN(left(\bar{j}), j)\}$.

2. If $\bar{i} + 1 < \bar{j}$, then calculate $MIN(right(\bar{i}) + 1, left(\bar{j}) - 1)$, using Step 3 of the algorithm. Notice that $MIN(right(\bar{i}) + 1, left(\bar{j}) - 1)$ corresponds to $\min\{b_{\bar{i}+1}, \ldots, b_{\bar{j}-1}\}$. Thus, $MIN(i, j) = \min\{MIN(i, right(\bar{i})), MIN(right(\bar{i}) + 1, left(\bar{j}) - 1), MIN(left(\bar{j}), j)\}$.

The value of $MIN(right(\bar{i}) + 1, left(\bar{j}) - 1)$ is obtained using Step 3, as follows. Each processor calculates $w = \text{LCA}(b_{\bar{i}+1}, b_{\bar{j}-1})$ in constant time, then determines the level l_w and determines v and u, the left and right sons of w, respectively. Processor $\bar{i} + 1$ calculates $S'[l_w - 1]$ and sends this value to Processor 0. Processor $\bar{j} - 1$ calculates $P'[l_w - 1]$ and sends this value to Processor 0. Processor 0, finally, calculates the minimum of the received minima. In both cases, Processor 0 receives the minima of Processors $\bar{i}$ and $\bar{j}$.

Notice finally that the correctness of Algorithm 3 results from the observations presented in this section.

6. Experimental Results

The proposed algorithm has been implemented on a distributed memory parallel machine, the Parsytec PowerXplorer 16/32 Parallel Computing System, with 16 nodes under a two-dimensional topology. Each node consists of one PowerPC601 application processor and one T805 communication processor, and 32 MBytes of local memory. The implementation is done in PVM (parallel virtual machine). We use the interface PowerPVM that implements PVM 3.2. Compared to the implementation with the native system called PARIX, we observe very similar performance results. PVM is chosen for portability.

For the experiments we use the input size $n = 2^k$, with $k = 5, \ldots, 19$, and $p = 2^j$ processors, with $j = 0, \ldots, 4$. The inputs consist of integers randomly distributed among the processors. The input distribution is not important, since the number of operations required in the execution of the algorithm does not depend on how the input data is distributed among the processors.

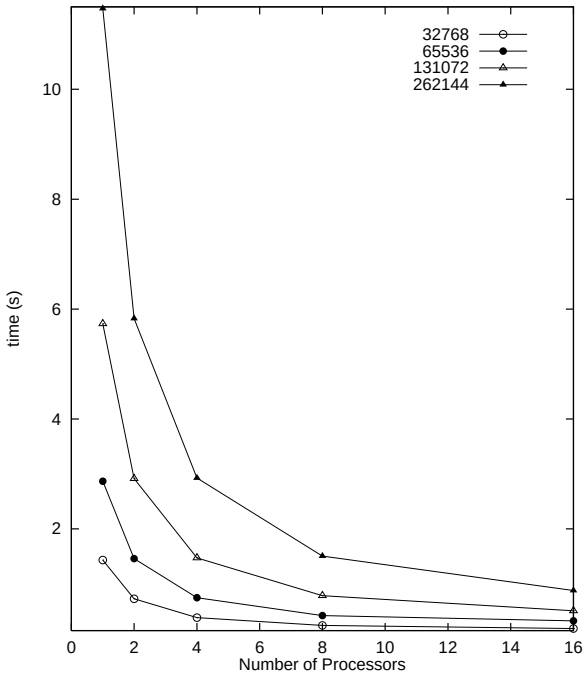

Fig. 4: Maximum times.

There is only one communication round in which the minima of the subarrays in each processor are broadcast to the others. Therefore, the amount of data exchanged depends on p and does not depend on n. Communication thus becomes less significant for large n.

We use the term *speedup* as the quotient between the sequential running time and the parallel running time. For our experiments, the sequential time is obtained

by using an optimized sequential algorithm (Gabow et al. [11]) running on a single processor, instead of running the parallel algorithm run on one processor.

For small n ($n < 2^{10}$), no significant gain is obtained due to communication overhead. The effect of communication becomes less influential for $n > 2^{10}$ and the speedup becomes significant for $n > 2^{15}$.

For a given experiment, with fixed n and p, we do not observe significant difference in the running times in different executions. Therefore, we run the experiments only once each to get the results.

Fig. 4 shows the running times (in seconds) for $n = 2^k$ with $k = 15, \ldots, 18$ and using 1, 2, 4, 8 and 16 processors.

Fig. 5 shows the speedup obtained. Notice the speedup curves approximate the optimal curve for large n.

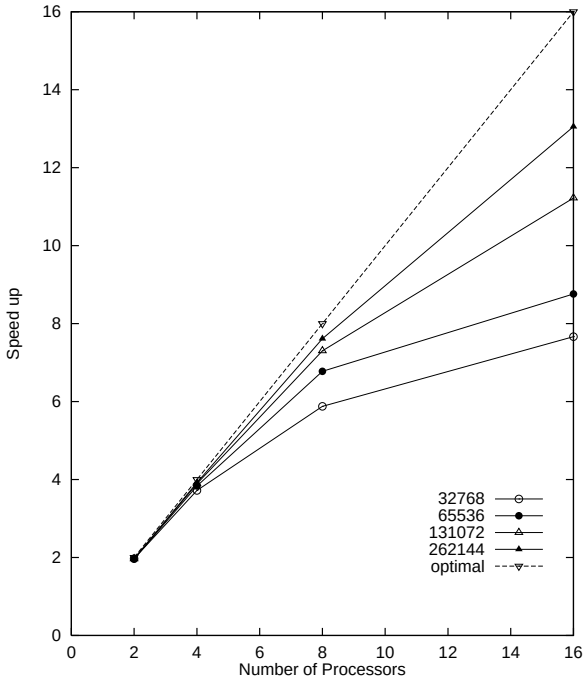

Fig. 5: Speed up.

7. Concluding Remarks

In the CGM(n, p) model, we are interested in designing parallel algorithms with linear local executing time and that utilize few communication rounds – preferably a constant number. This is because the communication between processors is more expensive than the computing time of the local data.

In this work we present an algorithm, in the CGM(n, p) model, for the range minima problem that runs in $O(n/p)$ time using $O(1)$ communication rounds. This

algorithm can be used to design a CGM algorithm for the LCA problem, based on the PRAM algorithm of JáJá [13], instead of other PRAM algorithms as in [14, 18], which would give less efficient implementations. Many graph algorithms in turn are based on the LCA problem.

In the design of the CGM algorithm, we chose to use the PS-tree to handle case 2(b) of Page 7 instead of merely constructing a Cartesian tree of an array of p numbers. This second approach, though feasible, would increase the local processing time. The small communication overhead, together with efficient local computing, enable us to obtain the final CGM algorithm, which is able to beat the very efficient sequential algorithm, as shown in the experimental results.

The experimental results are very promising, with speedup curves getting closer to the optimal speedup curve for large n, indicating the efficiency and scalability of the proposed algorithm. This is particularly significant once we know that this particular machine (Parsytec PowerXplorer) does not excel by any means in communication: The processors are in fact connected by Ethernet-like links. The efficiency of the proposed algorithm results from the constant number of communication rounds with low communication overhead and $O(n/p)$ computing time.

Acknowledgments

We want to thank the referees for their helpful comments.

References

1. A. V. Aho, J. E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley Publishing Company, 1975).
2. N. Alon and B. Schieber, "Optimal preprocessing for answering on-line product queries," Technical Report TR 71/87, The Moise and Frida Eskenasy Inst. of Computer Science, Tel Aviv University, 1987.
3. A. Amir, G. M. Landau, and U. Vishkin, "Efficient pattern matching with scaling," *Journal of Algorithms* **13** (1992) 2–32.
4. O. Berkman, B. Schieber, and U. Vishkin, "Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values," *Journal of Algorithms*, **14** (1993) 344–370.
5. O. Berkman and U. Vishkin, "Recursive star-tree parallel data structure," *SIAM J. Comput.*, **22(2)** (1993) 221–242.
6. E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song, "Efficient parallel graph algorithms for coarse grained multicomputers and BSP," in eds. A. Marchetti-Spaccamela, P. Degano, R. Gorrieri, *Proceedings ICALP '97 - 24th International Colloquium on Automata, Languages, and Programming*, (Springer-Verlag, Lecture Notes in Computer Science, vol. 1256, 1997) pp. 390–400.
7. F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar, "A randomized parallel 3d convex hull algorithm for coarse grained multicomputers," *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, 1995, pp. 27–33.
8. F. Dehne, A. Fabri, and C. Kenyon, "Scalable and architecture independent parallel geometric algorithms with high probability optimal time," *Proc. 6th IEEE*

Symposium on Parallel and Distributed Processing, 1994, pp. 586–593.

9. F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable parallel computational geometry for coarse grained multicomputers," *Proc. ACM 9th Annual Computational Geometry*, 1993, pp. 298–307.
10. F. Dehne and S. W. Song, "Randomized parallel list ranking for distributed memory multiprocessors," in eds. J. Jaffar and R. H. C. Yap, *Proceedings Asian Computing Science Conference*, (Springer-Verlag, Lecture Notes in Computer Science, vol. 1179, 1996) pp. 1–10.
11. H. N. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and related techniques for geometry problems," *Proc. 16th ACM Symp. On Theory of Computing*, 1984, pp. 135–143.
12. D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, **13(2)** (1984) 338–335.
13. J. Jájá, *An Introduction to Parallel Algorithms* (Addison-Wesley Publishing Company, 1992).
14. R. Lin and S. Olariu, "A simple optimal parallel algorithm to solve the lowest common ancestor problem," *Proc. of International Conference on Computing and Information - ICCI'91*, (Springer-Verlag, Lecture Notes in Computer Science, vol. 497, 1991) pp. 455–461.
15. H. Mongelli and S.W. Song, "A range minima parallel algorithm for coarse grained multicomputers," in eds. Jose Rolim et al. *Proceedings Irregular'99 - Sixth International Workshop on Solving Irregularly Structured Problems in Parallel*, San Juan, Puerto Rico, (Springer-Verlag, Lecture Notes in Computer Science, vol. 1586, 1999) pp. 1075–1084.
16. V. L. Ramachandran and U. Vishkin, "Efficient parallel triconectivity in logarithmic parallel time," *Proc. of AWOC'88*, (Springer-Verlag, Lecture Notes in Computer Science, vol. 319, 1988) pp. 33–42.
17. J. H. Reif, editor, *Synthesis of Parallel Algorithms* (Morgan Kaufmann Publishers, 1993).
18. B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization," *SIAM J. Comput.*, **17** (1988) 1253–1262.
19. L. G. Valiant, "A bridging model for parallel computation," *Comm. of the ACM*, **33** (1990) 103–111.
20. J. Vuillemin, "A unified look at data structures," *Comm. of the ACM*, **23** (1980) 229–239.