A BSP/CGM Algorithm for Finding All Maximal Contiguous Subsequences of a Sequence of Numbers*

C. E. R. Alves[1], E. N. Cáceres[2], and S. W. Song[3]

[1] Universidade São Judas Tadeu, São Paulo, SP, Brazil
prof.carlos_r_alves@usjt.br
[2] Universidade Fed. de Mato Grosso do Sul, Campo Grande, MS, Brazil
edson@dct.ufms.br
[3] Universidade de São Paulo, São Paulo, Brazil
song@ime.usp.br

Abstract. Given a sequence A of real numbers, we are interested in finding a list of all non-overlapping contiguous subsequences of A that are *maximal*. A *maximal* subsequence M of A has the property that no proper subsequence of M has a greater sum of values. Furthermore, M may not be contained properly within any subsequence of A with this property. This problem has several applications in Computational Biology and can be solved sequentially in linear time. We present a BSP/CGM algorithm that solves this problem using $p = O(|A|/p)$ processors in $O(|A|/p)$ time and $O(|A|/p)$ space per processor. The algorithm uses a constant number of communication rounds of size at most $O(|A|/p)$. Thus the algorithm achieves linear speed-up and is highly scalable. To our knowledge, there are no previous known parallel algorithms to solve this problem.

1 Introduction

Given a sequence of real numbers, the *maximum subsequence problem* consists of finding the contiguous subsequence with the maximum sum [3]. A more general problem is the *all maximal subsequences problem* [15] where we are interested in finding a list of all non-overlapping contiguous subsequences with maximal sum.

These two important problems arise in several contexts in Computational Biology. Many applications are presented in [15], for example, to identify transmembrane domains in proteins expressed as a sequence of amino acids and to discover CpG islands. Karlin and Brendel [10] define scores ranging from -5 to 3 to each of the 20 amino acids. For the human β_2-adrenergic receptor sequence, disjoint subsequences with the highest scores are obtained and these subsequences correspond to the known transmembrane domains of the receptor.

* Partially supported by FINEP-PRONEX-SAI Proc. 76.97.1022.00, CNPq Proc. 30.0317/02-6, 30.5218/03-4, 47.0163/03-8, 55.2028/02-9, and FUNDECT-MS Proc. 41/100117/03.

Csuros [5] mentions other applications that require the computation of such subsequences, in the analysis of protein and DNA sequences [4], determination of isochores in DNA sequences [9, 12], and gene identification [11].

Efficient linear time sequential algorithms are known to solve both problems [2, 3, 15]. Parallel solutions are known only for the basic maximum subsequence problem. For a given sequence of n numbers, Wen [18, 13] presents a EREW PRAM algorithm that takes $O(\log n)$ time using $O(n/\log n)$ processors. Qiu and Akl [14] developed a parallel algorithm for several interconnection networks such as the hypercube, star and pancake interconnection networks of size p. It takes $O(n/p + \log p)$ time with p processors. Alves, Cáceres and Song [1] present a BSP/CGM parallel algorithm on p processors that requires $O(n/p)$ computing time and constant number of communication rounds.

In this paper we present a BSP/CGM algorithm that solves the *all maximal subsequences* problem. To our knowledge, there are no previous known parallel algorithms to solve this problem. Given a sequence A of real numbers, the proposed algorithm uses p processors and finds all the maximal subsequences in $O(|A|/p)$ time, using $O(|A|/p)$ space per processor, and requiring a constant number of communication rounds. Unlike the parallel solution for the basic maximum subsequence problem, it is not at all intuitive that one can find a parallel algorithm for the all maximal subsequences problem that requires only a constant number of communication rounds in which at most $O(|A|/p)$ data are transmitted. In the following we present the main ideas and the approach utilized to derive the proposed algorithm.

2 Preliminary Definitions and Results

In part of this section we present the results of [15] for completeness. To design our parallel algorithm, we will see under which conditions the local maximal subsequences are potential candidates to be merged together to form larger maximal subsequences. Furthermore, we will present a modified and more detailed sequential algorithm to make this text as self-contained as possible. First we present some notation.

2.1 Notation

Consider a sequence A of real numbers. We denote the (whole) sequence of numbers by A and its elements by a_i, $1 \leq i \leq |A|$. Subsequences of A are indicated by their limits: $A_i^j = (a_{i+1}, ..., a_j)$. Notice that the superscript indicates the rightmost position in the subsequence, while the subscript is one less than the leftmost position. If the subscript and the superscript are equal, the subsequence is empty.

Sometimes a particular subsequence of A will be denoted by some other upper-case letter, but to avoid confusion all indices will refer to sequence A. To indicate the indices of the first (leftmost) and last (rightmost) positions of

a sequence X we use $L(X)$ and $R(X)$. For coherence with the previous paragraph we use $X = A_{L(X)}^{R(X)} = (a_{L(X)+1}, ..., a_{R(X)})$. Notice that $L(X)$ indicates one position to the left of the actual beginning of X.

The concatenation of sequences $X_1, X_2, ... X_n$ will be denoted by $\langle X_1, X_2, ... X_n \rangle$. Observe that a sequence X_i may consist of a single number.

The sum of the values of a subsequence X (the *score* of X) will be denoted by $Score(X)$. If X is empty, then we define its score to be zero. As the sum of prefixes of A is very important in this paper, we use $PS(j)$ to denote $Score\left(A_0^j\right)$. We consider $PS(0) = 0$. Notice that $Score\left(A_i^j\right) = PS(j) - PS(i)$. For a subsequence $X = A_i^j$, the minimum and the maximum among all values of $PS(k)$, for $i \le k \le j$, will be denoted by $Min(X)$ and $Max(X)$, respectively.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a_i	5	-3	-1	5	-9	0	3	3	7	-9	3	-6	3	-1	0	3	-3	0	7	-4	0	-6

Fig. 1. Example sequence to be used throughout the text.

2.2 Coarse Grained Multicomputer

For our parallel computation model, we use a simpler version of the BSP model [8, 17], referred to as the *Coarse Grained Multicomputer (CGM)* model [6, 7]. It is comprised of a set of p processors each $O(N/p)$ local memory, where N denotes the input size of the problem, and an arbitrary communication network. A CGM algorithm consists of alternating local computation and global communication rounds. In each communication round, each processor sends $O(N/p)$ data and receives $O(N/p)$ data. Finding an optimal algorithm in the CGM model is equivalent to minimizing the number of communication rounds as well as the total local computation time. Furthermore, it has been shown that this leads to improved portability across different parallel architectures [8, 17]).

2.3 Problem Definition

In this paper we only consider *contiguous* subsequences of a main sequence. We will omit the adjective for simplicity.

A *maximum* scoring subsequence of X is one whose score is the largest among all scores of subsequences of X. When ties occur, we choose the subsequence of minimum length and if a tie persists the choice of the particular subsequence is irrelevant. If there is no positive number in X, we consider that there is no maximum scoring subsequence.

With this definition, it is easy to see that prefixes and suffixes of a maximum subsequence always have positive scores, because the deletion of a prefix or suffix with non-positive score would lead to a better subsequence.

The problem of finding a maximum scoring subsequence of A is very well known and can be solved in linear time [3].

The problem of finding all maximal subsequences of A is more complicated. First, we need to define what is a maximal subsequence. Ruzzo and Tompa [15] define the set of maximal subsequences in a procedural way that can be put in the following recursive definition:

Definition 1. Set of maximal subsequences of a sequence A. *Given a sequence A of real numbers, the set of maximal subsequences of A is empty if A has no positive values. Otherwise, let $\langle A_1, M, A_2 \rangle$ be a decomposition of A in three subsequences where M is the maximum scoring subsequence of A (A_1 and A_2 may be empty sequences). Then the set of maximal subsequences of A is the union of $\{M\}$, the set of maximal subsequences of A_1 and the set of maximal subsequences of A_2.*

To facilitate the understanding of the main ideas in this paper, we consider the example sequence $A = (a_1, a_2, \ldots, a_{22})$ shown in Figure 1. The maximal subsequences are $A_0^4 = (5, -3, -1, 5)$, $A_6^9 = (3, 3, 7)$, $A_{10}^{11} = (3)$, and $A_{12}^{19} = (3, -1, 0, 3, -3, 0, 7)$, with respective scores of 6, 13, 3, and 9.

The definition above leads immediately to a recursive algorithm that in the worst case takes $O(n^2)$ time to find all maximal subsequences of a sequence of size n [4].

Ruzzo and Tompa also give two necessary and sufficient properties that a subsequence X must have to be maximal in sequence A. They are stated in the following theorem. For a proof, see [15].

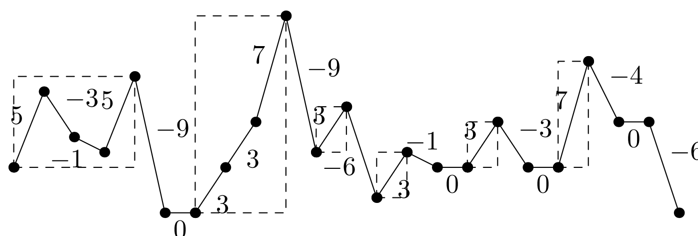

Fig. 2. Graphical representation of sequence $A = (5, -3, -1, \ldots, 0, -6)$. Some Pr1-subsequences are shown.

Theorem 1. *A subsequence X is maximal in A iff it has both properties below:*

Property Pr1 *For any proper subsequence Y of X, $Score(Y) < Score(X)$.*
Property Pr2 *There is no proper supersequence of X that has Property Pr1.*

[4] In the average case (given a reasonable definition of what is a random sequence of numbers), this algorithm will take $O(n \log n)$ time. This is not important in this paper.

Considering that the empty sequence is a subsequence of any other sequence, notice that the score of a sequence with property Pr1 must be positive. Subsequences of A that have property Pr1 will be called *Pr1-subsequences*.

We can restate the definition of a maximal subsequence in terms of these properties. We will use this new definition throughout the paper. The previous (equivalent) definition was presented because it is more natural when the applications are considered, while the new definition is better for understanding the linear algorithm of Ruzzo and Tompa and our parallel algorithm.

Definition 2. List of maximal subsequences of a sequence A**.** *Given a sequence A of real numbers, the list of maximal subsequences of A, denoted $MList(A)$, is the list of all subsequences that have Properties Pr1 and Pr2, ordered with respect to $L(.)$. This list is indexed starting at 1 with the leftmost subsequence.*

Property Pr1 can also be stated in terms of prefix sums.

Lemma 1. *A subsequence A_i^j is Pr1-subsequence iff for all m, $i < m < j$, $PS(i) < PS(m) < PS(j)$.*

Proof. If A_i^j is a Pr1-subsequence, $Score\left(A_i^j\right) > Score\left(A_i^m\right)$. Therefore $PS(j) - PS(i) > PS(m) - PS(i)$ and $PS(j) > PS(m)$. Also $Score\left(A_i^j\right) > Score\left(A_m^j\right)$ which leads to $PS(i) < PS(m)$.

If $PS(i) < PS(m) < PS(j)$ for all m, $i < m < j$, any A_l^r that is a proper subsequence of A_i^j has score $Score\left(A_l^r\right) = PS(r) - PS(l) < PS(j) - PS(i) = Score\left(A_i^j\right)$.

A graphical representation is useful here. We will plot the function $PS(.)$, so that positive (negative) values in the example sequence will be represented by ascending (descending) line segments (see Figure 2). A Pr1-subsequence X will be indicated by a rectangular box with $(L(X), PS(L(X)))$ and $(R(X), PS(R(X)))$ as lower-left and upper-right corners, respectively. The plotted curve touches the box only in these corners. Notice that the first three Pr1-subsequences in Figure 2 are maximal subsequences of A, but the last three are not (they are subsequences of the same A-maximal, namely A_{12}^{19}).

We say that A_i^j, $i < j$, is a *Pr1-prefix* if $PS(i) < Min\left(A_{i+1}^j\right)$ and it is a *Pr1-suffix* if $Max\left(A_i^{j-1}\right) < PS(j)$. A Pr1-subsequence is both a Pr1-prefix and a Pr1-suffix.

Corollary 1. *If P is a Pr1-prefix and S is a Pr1-suffix, $\langle P, S\rangle$ is a Pr1-subsequence iff $Min(P) < Min(S)$ and $Max(P) < Max(S)$.*

2.4 Some Results About Maximal Subsequences

We give some results that will be useful in the description of the sequential and the parallel algorithms that follow. First we present some lemmas from [15].

Lemma 2. *Any Pr1-subsequence of a sequence A is contained in a maximal subsequence of A (maybe not properly).*

Proof. Suppose the affirmation is not true. Let X be the largest Pr1-subsequence of A not contained in any maximal subsequence. Then X is not maximal, has not property Pr2 and therefore it must be contained in a larger Pr1-subsequence of A, which leads to a contradiction.

Lemma 3. *Given a sequence A, any two distinct maximal subsequences of A do not overlap or touch each other.*

Proof. Suppose this assertion is not true. Let A_i^k and A_j^l be two distinct maximal subsequences that violate the assertion. One maximal subsequence cannot be properly contained in another (property Pr2) so without loss of generality we may consider $i < j \leq k < l$. By Lemma 1 applied to A_i^k we have $PS(i) < PS(j)$ and the same lemma applied to both subsequences shows that $PS(i) < PS(m)$ for all m, $i < m < l$. Similarly, we can prove that $PS(m) < PS(l)$ for all m, $i < m < l$. Applying Lemma 1 in the other direction we conclude that A_i^l is a Pr1-subsequence, so A_i^k and A_j^l do not have property Pr2, a contradiction.

Both the sequential algorithm and the new parallel algorithm are based on finding lists of maximal subsequences in segments of the original sequence A. Consider a subsequence X of A. We will say that a subsequence is an *X-maximal subsequence*, or just an *X-maximal*, if it is maximal in X, that is, it is a Pr1-subsequence and has no proper supersequence that is a Pr1-subsequence of X. We want to find the set of all A-maximals.

Based on the previous lemma, we will say that an A-maximal is to the left of another if its $L(.)$ is smaller.

We will apply the previous lemmas to any subsequence of A, not only to A itself.

Lemma 4. *Let $Z = \langle X, Y \rangle$ for some non-empty X and Y. Then there is at most one Z-maximal M that overlaps both X and Y. If there is such M, it has an X-maximal as a prefix and a Y-maximal as a suffix. The X-maximals to the left of M and the Y-maximals to the right of M are also Z-maximals.*

Proof. By applying Lemma 3 to Z, it is obvious that no more than one Z-maximal overlaps X and Y. Let us suppose that there is such a Z-maximal M. $X = A_i^j$, $Y = A_j^k$ and $M = A_{m_1}^{m_2}$ for some $0 \leq i \leq m_1 < j < m_2 \leq k \leq |A|$. We now prove the affirmations concerning X. The affirmations concerning Y are proved analogously.

$PS(m_1)$ is the minimum prefix sum in M. If n is the smallest value in $]m_1, j]$ such that $PS(n)$ is maximum, then $A_{m_1}^n$ is a Pr1-subsequence of X. By Lemma 2, $A_{m_1}^n$ must be contained in an X-maximal. This X-maximal is a Pr1-subsequence of Z, so it must be contained in a Z-maximal, which can only be M. For this reason and the choice of n, we see that $A_{m_1}^n$ is an X-maximal that is a prefix of M.

Any X-maximal that is to the left of M is a Pr1-subsequence. If it is not a Z-maximal, then there is a Pr1-subsequence of Z that contains it, and the maximality in X forbids this larger Pr1-subsequence to be contained in X. So this Pr1-subsequence overlaps X and Y, contradicting the uniqueness of M.

The previous lemma is important for the sequential algorithm because it shows that it is possible to build $\mathit{MList}(A)$ working incrementally. Having a prefix X of A and its maximal subsequences, we can extend this prefix to the right, preserving some of the X-maximals and eventually creating another maximal subsequence that involves the extension and the rightmost X-maximals. The sequential algorithm appends just one number to X at each step, so $|Y| = 1$. We will show the details shortly.

Lemma 4 is also important for the parallel algorithm to be presented. Sequence A is divided into subsequences that are treated separately. Their maximal subsequences are used later to find the A-maximals.

The parallel algorithm deals with the following subproblem: given a subsequence X of A and its list of maximal subsequences $\mathit{MList}(X)$, find, if possible, an X-maximal that is a prefix (or suffix) of a larger A-maximal. This clearly involves $\mathit{MList}(X)$ and the rest of sequence A. However, some X-maximals need not be considered as possible prefixes or suffixes of larger A-maximals, regardless of what is outside X. The efficiency of our algorithm is based on this important notion, so we formalize it in the following definitions and lemmas. We deal with prefix candidates first.

Definition 3 **($\mathit{PList}(X)$).** *Given a subsequence X of A, $\mathit{PList}(X)$ is the ordered list of all X-maximals, with the exception of those X-maximals M for which one of the two conditions below are satisfied.*

1. $\mathit{Min}(M) \geq \mathit{PS}(R(X))$ *or*
2. *there is an X-maximal N to the right of M such that $\mathit{Min}(M) \geq \mathit{Min}(N)$.*

The elements of $\mathit{PList}(X)$ are indexed starting at 1 with the leftmost subsequence.

Informally, $\mathit{PList}(X)$ gives us the list of all X-maximals that are potential candidates to be merged to the right to give larger maximals. Notice that we excluded from $\mathit{PList}(X)$ those X-maximals (satisfying conditions 1 and 2) that can never give larger maximals. Consider $X = A_0^{14}$ of the example sequence (see Figure 1 and Figure 2). There are four X-maximals, namely A_0^4, A_6^9, A_{10}^{11}, and A_{12}^{13} (indicated by the first four boxes of Figure 2). A_0^4 does not belong to $\mathit{PList}(X)$ because of condition 1. A_{10}^{11} does not belong to $\mathit{PList}(X)$ because of both conditions 1 and 2. Thus $\mathit{PList}(X) = (A_6^9, A_{12}^{13})$.

Lemma 5. *If X is a subsequence of A, $\mathit{PList}(X)$ contains all X-maximals that can be a proper prefix of an A-maximal.*

Proof. For an X-maximal M, any of the two conditions in Definition 3 implies the existence of $i \in \,]L(M), R(X)]$ such that $\mathit{PS}(L(M)) \geq \mathit{PS}(i)$, so no A-maximal may extend from M past $R(X)$, because it would violate property Pr1.

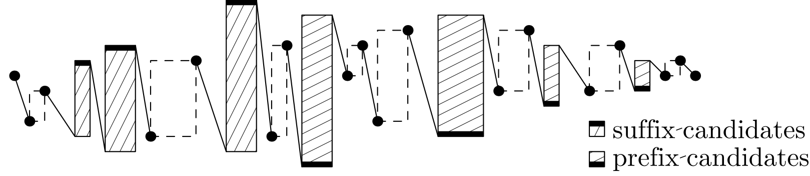

Fig. 3. Graphical representation of a sequence X, $\mathit{MList}\,(X)$, $\mathit{PList}\,(X)$ and $\mathit{SList}\,(X)$. The first (last) maximal is not a suffix (prefix) candidate because of the first condition of the definition. The other maximals that are not candidates fall in the second condition - observe the bottom of the prefix candidates and the top of the suffix candidates. The descending lines represent sequences of non-positive numbers.

Therefore, all X-maximals removed from $\mathit{PList}\,(X)$ are not proper prefixes of any A-maximal.

Lemma 6. *If M is a sequence in $\mathit{PList}\,(X)$ and $i \in \left]L(M), R(X)\right]$ then $\mathit{Min}\,(M) < \mathit{PS}\,(i)$, that is, $A_{L(M)}^{R(X)}$ is a Pr1-prefix.*

Proof. Suppose that it is possible to find an i in the specified range such that $\mathit{PS}\,(L(M)) \geq \mathit{PS}\,(i)$. Pick the largest possible i. Condition 1 of Definition 3 would forbid M in $\mathit{PList}\,(X)$ if $i = R(X)$, so $i < R(X)$. The choice of i guarantees that $\mathit{PS}\,(i+1) > \mathit{PS}\,(i)$, so A_i^{i+1} is a Pr1-sequence and it must be contained in some X-maximal N (Lemma 2). N has to be to the right of M and $\mathit{Min}\,(M) \geq \mathit{PS}\,(i) \geq \mathit{Min}\,(N)$, but then Condition 2 of Definition 3 would also forbid M in $\mathit{PList}\,(X)$, a contradiction.

Lemma 7. *If M is a sequence in $\mathit{PList}\,(X)$ and $i \in \left]R(M), R(X)\right]$ then $\mathit{Max}\,(M) \geq \mathit{PS}\,(i)$.*

Proof. Suppose that it is possible to find an i in the specified range such that $\mathit{Max}\,(M) < \mathit{PS}\,(i)$. Pick the smallest possible i. Pick the largest value of j such that $L(M) \leq j < i$ and $\mathit{PS}\,(j)$ is minimum in the range. It is clear by Lemma 1 that A_j^i has property Pr1, so there is an X-maximal N that contains it. As distinct X-maximals cannot overlap (Lemma 3), N must be to the right of M. By the choice of j we must have $\mathit{Min}\,(N) \leq \mathit{Min}\,(M)$, but then M should not be in $\mathit{PList}\,(X)$ by condition 2 of Definition 3, a contradiction.

A direct consequence of the previous lemmas is that $\mathit{PList}\,(X)$ is in a non-increasing order of $\mathit{Max}\,(.)$ and a strictly increasing order of $\mathit{Min}\,(.)$. Figure 3 illustrates $\mathit{PList}\,(X)$ (and $\mathit{SList}\,(X)$, defined shortly).

For the parallel algorithm we will a need similar definition for possible suffixes of A-maximals. The definition and associated lemmas are given now (without proofs, which are similar to the previous ones). Notice the exchanging roles of $\mathit{Max}\,(.)$ and $\mathit{Min}\,(.)$, “left” and “right”, etc.

Definition 4 ($SList(X)$)**.** *Given a subsequence X of A, $SList(X)$ is an ordered list of all X-maximals, with the exception of those X-maximals N for which one of the two conditions below are satisfied.*

1. $Max(N) \leq PS(L(X))$ *or*
2. *there is a X-maximal M to the left of N such that $Max(N) \leq Max(M)$.*

The elements of $SList(X)$ are indexed starting at 1 with the rightmost subsequence.

Lemma 8. *If X is a subsequence of A, $SList(X)$ contains all X-maximals that can be a proper suffix of an A-maximal.*

Lemma 9. *If N is a sequence in $SList(X)$ and $i \in [L(X), R(N)[$ then $PS(i) < Max(N)$, that is, $A_{L(X)}^{R(N)}$ is a Pr1-suffix.*

Lemma 10. *If N is a sequence in $SList(X)$ and $i \in [L(X), L(N)[$ then $PS(i) \geq Min(N)$.*

Notice that at most one X-maximal may belong to both $PList(X)$ and $SList(X)$, namely the maximum subsequence of X. Any other element of $SList(X)$ must be to the left of any element of $PList(X)$. See Figure 3 for an illustration of $PList(X)$ and $SList(X)$ (when these lists are disjoint).

2.5 The Sequential Algorithm

We now present Algorithm 1, a modified version of the sequential algorithm of Ruzzo and Tompa [15]. There are several differences of Algorithm 1 from the original version of [15]. We present the procedure in a more explicit way, making use of arrays to facilitate the analysis and using one less level of loop nesting, but the main ideas and the performance are the same. We present this algorithm for completeness, as the sequential algorithm is also used in the parallel one. We also want to make explicit the construction of $PList(.)$, which is implicitly used in the original algorithm of Ruzzo and Tompa as an auxiliary linked list.

The input of the algorithm is the sequence A and the output is $MList(A)$ (*Ml* for short in the algorithm) and $PList(A)$ (*Pl* for short). Both lists are implemented as arrays with first index 1 and used as *stacks* with index 1 referring to the bottom. *Pl* will actually store indices of elements in *Ml* while the latter will store the data about the A-maximals ($L(.)$, $R(.)$, $Max(.)$ and $Min(.)$).

Theorem 2. *Given a numerical sequence A, Algorithm 1 computes $MList(A)$ and $PList(A)$ correctly using $O(|A|)$ time and space.*

Proof. We will prove that at the end of each iteration of the loop in line 1 *Ml* and *Pl* represents $MList\left(A_0^i\right)$ and $PList\left(A_0^i\right)$, respectively. Notice that in the beginning of the first iteration we have $i = 1$ and A_0^0 is the empty subsequence. Both *Ml* and *Pl* are empty, representing $MList\left(A_0^0\right)$ and $PList\left(A_0^0\right)$.

Algorithm 1 Maximal Subsequences (Sequential)

Require: Sequence $A = (a_1, a_2, \ldots, a_{|A|})$

Ensure: Arrays Ml and Pl, with n_m and n_p elements, respectively. s keeps the prefix sum.

```
 1: n_m ← 0, n_p ← 0, s ← 0
 2: for i ← 1 to |A| do
 3:   s ← s + a_i
 4:   if a_i < 0 then
 5:     while n_p > 0 and Min(Ml[Pl[n_p]]) ≥ s do
 6:       n_p ← n_p − 1 {Pop prefix candidates}
 7:     end while
 8:   end if
 9:   if a_i > 0 then
10:     {Push new sequence formed by a_i only (partial data, may be discarded)}
11:     n_m ← n_m + 1
12:     Min(Ml[n_m]) ← s − a_i {Previous s}
13:     L(Ml[n_m]) ← i − 1
14:     Pl[n_p + 1] ← n_m
15:     while n_p > 0 and Max(Ml[Pl[n_p]]) < s do
16:       n_p ← n_p − 1 {Pop prefix candidates, looking for the best to merge with a_i}
17:     end while
18:     n_p ← n_p + 1
19:     {Pl[n_p] is the best prefix candidate}
20:     n_m ← Pl[n_p] {Pop sequences}
21:     {Complete the data of the top sequence}
22:     R(Ml[n_m]) ← s
23:     Max(Ml[n_m]) ← i
24:   end if
25: end for
```

Consider $X = A_0^{i-1}$ and $Z = A_0^i$. We now show that the body of the loop builds $MList(Z)$ and $PList(Z)$ based on $MList(X)$ and $PList(X)$. After line 1, $s = PS(R(Z))$ and $s - a_i = PS(R(X))$.

Using Lemma 4, we search for a unique Z-maximal that overlaps X and ends in a_i. If a_i is non-positive, then by Lemma 1 it cannot be the suffix of any maximal, so $MList(Z) = MList(X)$. Based on this, $PList(Z)$ should be the same as $PList(X)$, except for the X-maximals that must be removed according to the first condition in Definition 3. So the loop in line 1 removes all Z-maximals that have $Min(.)$ not less than $PS(R(Z))$.

If $a_i > 0$ then it must be included in some Z-maximal. Lines 1 through 1 introduce a new sequence, containing only a_i, in Ml and Pl. This sequence is not necessarily a Z-maximal. Only the data that refer to the beginning of this sequence ($L(.)$ and $Min(.)$) are introduced in Ml. Now the algorithm tries to find the largest possible Pr1-subsequence of Z that contains a_i, that is, a Z-maximal.

Applying Lemmas 4 and 5 to Z, the possible prefixes for this Z-maximal are the elements of $PList(X)$ (or a_i itself). By Lemma 6, if M is a sequence in $PList(X)$ then $A_{L(M)}^{i-1}$ is a Pr1-prefix. The sequence formed by a_i alone is a Pr1-suffix, so we may apply Corollary 1: $A_{L(M)}^{i}$ is a Pr1-subsequence if and only if $Min(M) < PS(i-1)$ and $Max(M) < PS(i)$. The first inequality holds by Definition 3 (see Condition 1). The second inequality requires a search in $PList(X)$.

By Lemma 7, if an element M of $PList(X)$ satisfies the second inequality, all elements to the right of M also satisfy it. We are interested in the leftmost element of $PList(X)$ that satisfies the inequality, for it leads to the largest possible Pr1-sequence. The loop in line 1 searches for this sequence. Once it is found, the data related to its termination ($R(.)$ and $Max(.)$) is changed to reflect the extension of the sequence up to a_i (lines 1 and 1). All sequences in $MList(X)$ from M to the end of $MList(X)$ are discarded and substituted by the new sequence (line 1) and the sequences to the left of M are maintained, in accordance to Lemma 4.

Finally, notice that the algorithm removes from Pl just the sequences that were absorbed by the new one, which is still in this array. By Definition 3, there is no reason to remove any of the other sequences, because no new sequence with smaller $Min(.)$ was introduced and $PS(Z)$ is larger than $PS(X)$ ($a_i > 0$), so we end up with $Pl = PList(Z)$.

Notice that the loop in line 1 may fail in the first test, indicating that no element of $PList(X)$ may be a proper prefix of a Z-maximal. In this case, the sequence introduced in lines 1 through 1 is used as M in the previous paragraph. $PList(Z)$ is equal to $PList(X)$ with the inclusion of this last sequence. No other sequence needs to be eliminated.

We now prove that the algorithm uses only $O(|A|)$ time and space. Ml and Pl will have approximately $|A|/2$ elements in the worst case, so the linearity of space is clear. The main loop in line 1 runs $|A|$ iterations. Every command in this loop clearly runs in constant time, except the loops in lines 1 and 1. But using amortized analysis, observing that n_p never becomes negative. It is clear that

the total number of iterations of these two loops (that is, the number of times that n_p is decremented) is limited by the number of times n_p is incremented in line 1, which is $O(|A|)$. We conclude that the algorithm runs correctly using $O(|A|)$ space and time.

3 The Parallel Algorithm

We now present the CGM algorithm to find all maximal subsequences of a sequence A using p processors, named P_i, $i \in [1, p]$. We assume that A is divided into p subsequences, each of size $l = \lceil |A|/p \rceil$ except the last one, which may be smaller. We call these subsequences $AP_i = A_{l(i-1)}^{li}$.

At the beginning of the procedure, for all $i \in [1, p]$ AP_i is already stored in the local memory of processor P_i. At the end, processor P_i will contain the information (position and score) of all A-maximals that start or end within AP_i.

3.1 Finding the Local Maximals

The results of Section 2.5 allow us to state the following:

Lemma 11. *In $O(|A|/p)$ time and space and using one communication round of size $O(p)$, each processor P_i ($i \in [1, p]$) may acquire the following information:*

- *its local lists of maximals ($MList\,(AP_i)$), prefix candidates ($PList\,(AP_i)$) and suffix candidates ($SList\,(AP_i)$).*
- *$PS\,(L\,(AP_j))$, $Min\,(AP_j)$ and $Max\,(AP_j)$ for* all $j \in [1, p]$.

Proof. When run by processor P_i, Algorithm 1 gives $MList\,(AP_i)$, $PList\,(AP_i)$ and $Score\,(AP_i)$, but without the information from the other processors it has to suppose that $PS\,(L\,(AP_i)) = 0$. The actual value is not important for the construction of the lists, but it must be added later to the values of the prefix sums in these lists.

Using Definition 4, a simple scan through $MList\,(AP_i)$ gives $SList\,(AP_i)$. This scan allows the obtention of $Min\,(AP_i) - PS\,(L\,(AP_i))$ and $Max\,(AP_i) - PS\,(L\,(AP_i))$.

The last two values and $Score\,(AP_i)$ can be broadcasted to all processors in one communication round of size $O(p)$. All processors will have $Score\,(AP_j)$ for all $j \in [1, p]$ and will be able to calculate $PS\,(L\,(AP_j))$, $Min\,(AP_j)$ and $Max\,(AP_j)$ for all $j \in [1, p]$. This may seem inefficient, but under our considerations it is better than parallelizing this simple operation and spending more time in communication.

Each processor can then update the values of the prefix sums in its three lists of results. It is easy to see that all the operations described here can be done in $O(|A|/p)$ time and space.

3.2 Basic Procedure for Joining Lists of Maximals

We will now see how $MList\,(Z)$ may be obtained from $MList\,(X)$, $MList\,(Y)$, $PList\,(X)$ and $SList\,(Y)$ when $Z = \langle X, Y \rangle$. The procedure shown here is the basis for our parallel algorithm, but the reader must know that the algorithm is not based on successive steps of pairwise joining of subsequences. Such a strategy would lead to $O(\log p)$ rounds of communication and ultimately to a sublinear speed-up. Later we will show how the partial data described in Section 3.1 are used in a global joining operation.

The following lemma states the condition for two local maximal subsequences to be merged to form a larger one.

Lemma 12. *Given $M \in PList\,(X)$ and $N \in SList\,(Y)$, $A_{L(M)}^{R(N)}$ is a Pr1-subsequence iff $Min\,(M) < Min\,(N)$ and $Max\,(M) < Max\,(N)$.*

Proof. Let $m = L\,(M), l = R\,(X) = L\,(Y), n = R\,(N)$. Lemmas 6 and 9 establish that A_m^l and A_l^n are respectively a Pr1-prefix and a Pr1-suffix. Lemmas 7 and 10, along with Lemma 1 applied to M and N, establish that $Max\,(M) = Max\left(A_m^l\right)$ and $Min\,(N) = Min\,(A_l^n)$. The lemma follows from Corollary 1 applied to $A_m^n = \langle A_m^l, A_l^n \rangle$.

Lemmas 5 and 8 state that we may search for a Z-maximal that overlaps X and Y using only $PList\,(X)$ and $SList\,(Y)$. Algorithm 2 does this. We use $Pl = PList\,(X)$ and $Sl = SList\,(Y)$ for short, indexing them as stated in Definitions 3 and 4. The algorithm returns the indices of the chosen candidates for prefix and suffix of the new Z-maximal. In this algorithm we use the elements of Pl and Sl of actual sequences, not as indices to lists of maximals, for simplicity.

Algorithm 2 Joining Two Lists of Maximals

Require: Lists Pl and Sl, with $|Pl|$ and $|Sl|$ candidates, respectively.
Ensure: Flag f that indicates if a new maximal was found, indices i_p and i_s of the candidates that define this maximal.

```
 1: i_p ← 1, i_s ← 1, f ← false
 2: while i_p ≤ |Pl| and i_s ≤ |Sl| and not f do
 3:   if Max (Pl[i_p]) ≥ Max (Sl[i_s]) then
 4:     i_p ← i_p + 1
 5:   else if Min (Pl[i_p]) ≥ Min (Sl[i_s]) then
 6:     i_s ← i_s + 1
 7:   else
 8:     f ← true
 9:   end if
10: end while
```

Lemma 13. *Given $Z = \langle X, Y \rangle$, $Pl = PList\,(X)$ and $Sl = SList\,(Y)$, Algorithm 2 finds the only Z-maximal that overlaps X and Y, if it exists, in $O(|Pl| + |Sl|)$ time and $O(1)$ additional space.*

Proof. The time and space complexity of Algorithm 2 is clearly as stated. We need to prove that it actually finds the Z-maximal, if it exists. Recall that, by Lemma 4, this Z-maximal is unique.

We now prove by induction the following affirmation: at the moment the loop test is performed, no Z-maximal exists with prefix $Pl[i]$ with $i \in [1, i_p[$ or with suffix $S[j]$ with $j \in [1, i_s[$.

The affirmation is clearly true for the first test, as there is no prefix or suffix candidates in the specified ranges. Suppose the affirmation is true for a particular iteration. The conditional statements inside the loop will perform as follows: if the first test results *true*, then there is no remaining suffix candidate in Sl with $Max(.)$ greater than $Max(Pl[i_p])$ (Lemma 9). By Lemmas 8 and 12 and the induction hypothesis we conclude that $Pl[i_p]$ is not a proper prefix of any valid Pr1-subsequence of Z, so i_p is increased and the affirmation remains true. The analysis for the case when the second test results *true* is similar.

If the loop ends with $f = false$ then there is no new Z-maximal. If the loop ends with $f = true$ then $Pl[i_p]$ and $Sl[i_s]$ satisfy the conditions of Lemma 12 and thus define a Pr1-subsequence of Z. The affirmation just proved shows that there is no other Pr1-subsequence that may properly contain the one defined by $Pl[i_p]$ and $Sl[i_s]$, so this subsequence has property Pr2 and is a Z-maximal.

3.3 Tagging the Local Candidates

The parallel algorithm performs a single joining step, using a constant number of communication rounds, involving all the local maximals found in the local step. This step is based on the simple observation that a non-local maximal must start inside some AP_i and end in some AP_j with $1 \leq i < j \leq p$, so it must have some sequence in $PList(AP_i)$ as prefix and some sequence in $SList(AP_j)$ as suffix.

The problem is to find a *relevant* set of Pr1-subsequences of A that cross processor boundaries. By *relevant* we mean that all the A-maximals that cross processor boundaries must be contained in this set. In a last step we just have to choose the Pr1-subsequences that are not contained in another one.

We say that a prefix candidate and a suffix candidate *match* if they define a Pr1-subsequence of A. The following definition states the conditions for a match.

Lemma 14. *For $M \in PList(AP_i)$ and $N \in SList(AP)_j$, $1 \leq i < j \leq p$, $A_{L(M)}^{R(N)}$ (the sequence that has M as prefix, N as suffix and contains AP_k, $i < k < j$) is a Pr1-subsequence iff $Min(M) < Min(N)$, $Max(M) < Max(N)$, $Min(M) < \min_{i<k<j} Min(AP_k)$ and $Max(N) > \max_{i<k<j} Max(AP_k)$.*

Proof. The proof is very similar to the proof of Lemma 12. The extra conditions involving AP_k, $i < k < j$, are related to Lemma 1.

After the local step described in Section 3.1 the processors cannot determine which candidates match because they have access only to their own lists of candidates. However, given a particular prefix or suffix candidate, the extra conditions exposed in Lemma 14 allow the determination of the processors where

a match for this candidate may be found. So, the first step in the global joining operation is to *tag* each candidate with the number of the processor(s) that may contain a match for it. We will see that each candidate receives at most one tag, with few exceptions.

Let us consider how to tag the prefix candidates of AP_i (the case for suffix candidates is similar). To simplify the discussion, we will consider a list of possible tags for these prefixes, named $PTagList\,(i)$. This list is constructed as follows. Initially, $PTagList\,(i)$ will contain one element for each processor with number greater than i. For processor $j \in\,]i,p]$ an element T will contain the following information: the number of the processor represented $tag(T) = j$ (the tag itself), the maximum and the minimum of the prefix sums inside the associated subsequence of A, $Max\,(T) = Max\,(AP_j)$ and $Min\,(T) = Min\,(AP_j)$. T will also contain the minimum of prefix sums in *all* local sequences from AP_{i+1} to AP_{j-1}: $Min{*}\,(T) = \min_{i<k<j} Min\,(AP_k)$ (∞ if $j = i+1$). $Min{*}\,(T)$ will be useful in the search for tags in accordance to Lemma 14. The whole list of tags is easily built in $O(p)$ time.

From this list we eliminate the elements that have a tag k such that there is j, $i < j < k$ and $Max\,(AP_j) \geq Max\,(AP_k)$. This is because any suffix candidate $N \in SList\,(AP_k)$ would have $Max\,(N) \leq Max\,(AP_k) \leq Max\,(AP_j)$, not being matchable with any prefix candidate of AP_i (Lemma 14). The elimination of all sequences in this condition takes $O(p)$ time.

The final list is indexed in descending order according to *tag*, starting with index 1. This indexing is used to make $PTagList\,(i)$ similar to a list of suffix candidates. The tagging algorithm, to be presented shortly, is very similar to Algorithm 2.

Based on all that was stated we may claim the following:

Affirmation 1 *For any $i \in [1,p]$, $PTagList\,(i)$ contains* all *the tags that represent local sequences that may have a suffix candidate that matches some prefix candidate of AP_i. The indexing of $PTagList\,(i)$ puts it in a decreasing order of $tag(.)$, a (strictly) decreasing order of $Max\,(.)$ and a non-decreasing order of $Min{*}\,(.)$.*

Let us now consider the tagging of a particular prefix candidate M.

Observation 1 *When comparing M with some $T \in PTagList\,(i)$, the following cases may occur:*

Case 1. $Max\,(M) \geq Max\,(T)$. This disqualifies $tag(T)$ for M (Lemma 14). As $PTagList\,(i)$ is ordered in a decreasing order of $Max\,(.)$, the following tags in $PTagList\,(i)$ are also disqualified.

Case 2. $Max\,(M) < Max\,(T)$. This opens two subcases:

Case 2.i *$Min\,(M) \geq Min{*}\,(T)$. This disqualifies $tag(T)$ for M (again by Lemma 14). As $PList\,(AP_i)$ is ordered in an increasing order of $Min\,(.)$ (Lemma 6), $tag(T)$ is also disqualified for the following prefix candidates of AP_i.*

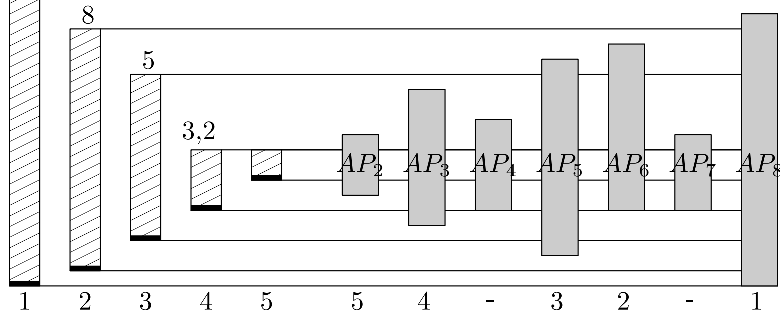

Fig. 4. Graphical representation of the results of the tagging procedure. We consider the tagging of elements of $PList\,(AP_1)$, represented as shaded bars on the left. The darkened bars in the right represent the data from other processors. The numbers below the bars represent the indices in $PList\,(AP_i)$ and $PTagList\,(1)$ (when applicable). The first prefix candidate has no tags due to case 1 of Observation 1. The second receives tag 8 based on case case 2.ii. The third rejects tags 6 and 8 based on case 2.i, and is tagged 5 based on case 2.ii. The fourth receives two tags and blocks the tagging of the fifth prefix candidate based on case 2.ii.

Case 2.ii *$Min\,(M) < Min{*}\,(T)$. There is a possibility of a match, so $tag(T)$ is a valid tag for M. Moreover, if $Min\,(M) < Min\,(T)$ then it is assured that there will be a match between M and a suffix candidate in $AP_{tag(T)}$, (namely N with $Max\,(N) = Max\,(T)$). This disqualifies all the following tags in $PTagList\,(i)$ because they would lead to sequences that cannot be maximals, since they would overlap the Pr1-subsequence that is sure to exist. Also, prefix candidates following M in $PList\,(AP_i)$ will lead to sequences that cannot be A-maximals, so they do not need to be tagged.*

Figure 4 illustrates the tagging of prefix candidates and exemplifies the three cases above. Algorithm 3 contains the tagging procedure for the prefix candidates of $PList\,(AP_i)$, called Pl for short. $PTagList\,(i)$ is called Tl for short and is preprocessed in accordance to Affirmation 1.

Lemma 15. *For $i \in [1, p]$ it is possible to tag all the elements of $PList\,(AP_i)$ and $SList\,(AP_i)$ based on the values of $Max\,(AP_j)$ and $Min\,(AP_j)$ for all $j \in [1, p]$. Each tag indicates which processor may contain a match for a particular candidate. Each candidate is tagged at most once, with two exceptions per processor at the most. The time required is $O(|A|/p)$ and the space required is $O(p)$.*

Proof. The proof is based on Algorithm 3 and Observation 1. We first prove that Algorithm 3 tags correctly all prefix candidates by proving the following invariant affirmation: at the moment the loop test will be performed, all elements of $PList\,(AP_i)$ with index less than i_p received all the tags it can get and all elements of $PTagList\,(i)$ with index less than i_t were used to tag all the candidates they can.

The affirmation is obviously true at the first time the test is reached. Supposing it is true at the beginning of an iteration of the loop, three cases may occur:

– line 3 is executed, because $Max\,(Pl[i_p]) \geq Max\,(Tl[i_t])$ and case 1 applies. No more tags may be applied to $Pl[i_p]$, so i_p is incremented and the invariant remains true.
– line 3 is executed. Case 2.i applies, so $Tl[i_t]$ cannot be used to tag any prefix candidate, allowing i_t to be incremented.
– line 3 is executed. Case 2.ii applies and $Pl[i_p]$ is tagged with $tag(Tl[i_t])$. Furthermore, if $Min\,(Pl[i_p]) < Min\,(Tl[i_t])$ then, still due to case 2.ii, the loop is ended and no more tagging is done.

Algorithm 3 Tagging a List of Prefix Candidates

Require: Lists Pl and Tl, with $|Pl|$ and $|Tl|$ elements, respectively.
Ensure: Tagging of the elements of Pl.

```
 1: i_p ← 1, i_t ← 1, f ← false
 2: while i_p ≤ |Pl| and i_t ≤ |Tl| and not f do
 3:   if Max (Pl[i_p]) ≥ Max (Tl[i_t]) then
 4:     i_p ← i_p + 1
 5:   else if Min (Pl[i_p]) ≥ Min* (Tl[i_t]) then
 6:     i_t ← i_t + 1
 7:   else
 8:     tag Pl[i_p] with tag(Tl[i_t])
 9:     if Min (Pl[i_p]) < Min (Tl[i_t]) then
10:       f ← true
11:     end if
12:   end if
13: end while
```

Therefore, Algorithm 3 performs the tagging correctly. The time required is $O(|Pl| + |Tl|) = O(|A|/p + p) = O(|A|/p)$, including the time to build $PTagList\,(i)$. The space required is $O(p)$, for $PTagList\,(i)$. A similar procedure can be done to $SList\,(AP_i)$.

Now, suppose that a prefix candidate $Pl[i'_p]$ is tagged twice, based on $Tl[i'_t]$ and $Tl[i'_t+1]$ (The fact that the tags should be consecutive is easy to prove). The tagging occurs only at line 3. For the second tag, it is clear that $Min\left(Pl[i'_p]\right) < Min\,(Tl[i'_t+1])$, since the first tagging occurred because $Min\left(Pl[i'_p]\right) < Min*\,(Tl[i'_t]) \leq Min\,(Tl[i'_t+1])$. So, when a prefix candidate receives the second tag the loop stops. Something similar may occur to a suffix candidate, making for the second exception to the "one tag only" rule.

3.4 Finding Cross-Processors Pr1-subsequences

After the tagging procedure described in the previous section, each prefix/suffix candidate may be associated with two other processors: the one which contains

it and the one specified in the tag. Some candidates have no tags and may be ignored. A few candidates have two tags and have to be duplicated for the next phase.

The next phase involves checking the existence of cross-processors Pr1-subsequences of A, that is, Pr1-subsequences that start within AP_i and ends within AP_j for some pair (i, j), $1 \leq i < j \leq p$. This is done by checking the elements of $\mathit{PList}\,(AP_i)$ that are tagged with j and elements of $\mathit{SList}\,(AP_j)$ that are tagged with i. These elements must be in the local memory of one single processor for verification by Algorithm 2. The rule to choose which processor does the verification is simple: the one whose list of candidates is larger receives the data from the other one. In case both lists have the same size, a deterministic rule is used to break the tie. For example, if $i + j$ is even then P_i does the job, otherwise P_j does it.

This may be done for all pairs of processors using two communication rounds. In the first one, each processor P_i sends for processor $P_j \neq P_i$ the number of tags j that were used in $\mathit{PList}\,(AP_i)$ or $\mathit{SList}\,(AP_i)$. Each processor sends one number for each of the other processors, so the size of the communication round is $O(p)$. With these data, each pair of processors agree about which one shall receive the data from the other one. These data are sent in the next communication round.

Notice that each processor may send/receive at most the number of data present in its own lists of prefix/suffix candidates. Therefore, this communication round has size $O(|A|/p)$.

Each processor then searches for Pr1-subsequences that start or end within its local subsequence of A. Let us consider the search for the Pr1-subsequences that start within AP_i, $i \in [1, p[$. For each $j \in]i, p]$, processor P_i uses Algorithm 2 to find the largest Pr1-subsequence with ends in AP_i and AP_j (supposing that this processor, and not P_j, was selected to do the job). The prefix candidates are the elements of $\mathit{PList}\,(AP_i)$ that were tagged with j (let us say there are r of them) and the suffix candidates that were sent by processor P_j (s of them). By Lemma 13 the time taken is $O(r + s)$. This procedure must be repeated by processor P_i for all $j \in]i, p]$, taking a total time of $O(|\mathit{PList}\,(AP_i)\,|)$. A similar procedure must be done by P_i for all $j \in [1, i[$, taking $O(|\mathit{SList}\,(AP_i)\,|)$ total time. The whole procedure takes $O(|A|/p)$ time.

When looking for Pr1-subsequences that start within AP_i, processor P_i must first search for sequences that end in AP_p and proceed going down to AP_{i+1}. When a Pr1-subsequence is found the procedure may stop, because the next Pr1-subsequences would be contained in the first one. This means that each processor will find at most two new Pr1-subsequences, one ending and one starting in its local subsequence of A.

Now we may state the following lemma, already proved by the discussion above.

Lemma 16. *After tagging the prefix and suffix candidates as explained in Section 3.3, all cross-processors Pr1-subsequences that may be A-maximals can be found in $O(|A|/p)$ time and space and two communication rounds of sizes $O(p)$ and $O(|A|/p)$. The number of sequences is at most $2p$.*

It should be noticed that some of the new Pr1-subsequences may not have Property Pr2. The important thing here is that the procedure just described does not miss any possible A-maximal. The next step is finding the Pr1-subsequences that are really A-maximals.

3.5 Finding the new A-maximals

In the final step, all processors broadcast the information about the new Pr1-subsequences found. This involves a fourth communication round of size $O(p)$ (every processor sends at most two new subsequences and receives all of them). Every processor then eliminates the Pr1-subsequence that are contained in another Pr1-subsequence. This may seem redundant, but it is better to make all processors perform this computation than spend another communication round.

It should be noticed that the procedure described in the previous section does not generate two Pr1-subsequences that overlap, unless one is contained in the other. That is because if two Pr1-subsequences overlap then the union of them (that is, the subsequence of minimum length that contains both of them) is also a Pr1-subsequence (this is easily proved using Lemma 1). This union is larger than each of the two overlapping subsequences and so should have been detected by the procedure described earlier.

Each Pr1-subsequence is related to a different pair of processors. All that must be verified is which pairs generated new sequences. Algorithm 4 does this verification.

Lemma 17. *Algorithm 4 finds all cross-processors A-maximals based on the list of cross-processors Pr1-subsequences cited in Lemma 16. The time and space taken is $O(p)$.*

Proof. Lines 4 to 4 build array V in time $O(p+|L|) = O(p)$. $V[i]$, $i \in [1,p]$ stores the largest j for which there is a new Pr1-subsequence that starts within AP_i and ends within AP_j. If there is no such Pr1-subsequence, $V[i] = i$.

The following lines build the list of new A-maximals N. An invariant affirmation is that at line 4 all new A-maximals that start within AP_i for any $i \in [1,k[$ were already found. This is certainly true for the first time line 4 is reached. In the loop body, if the test in line 4 results *true* then a new A maximal is found. All Pr1-sequences that start within AP_i for $k < i < V[k]$ are contained in this new A-maximal and should be ignored, which is done in line 4. If the test results *false*, there is no new A-maximal that starts within AP_k, so k is incremented, keeping the invariant true.

A final step is done locally by each processor. By examining the list of new A-maximals, processor P_i may verify if there is an A-maximal that contains its entire local subsequence AP_i, which means that its own local set of maximals $MList\,(AP_i)$ should be discarded. This verification can be done in time $O(p)$. If there is a cross-processors A-maximal that starts or ends within AP_i, a final scan of $MList\,(AP_i)$ may eliminate the local maximals that are contained in a larger

Algorithm 4 Elimination of Pr1-subsequences that are not A-maximals

Require: List L (with $|L|$ elements) of pairs of processors for which there is a cross-processor Pr1-subsequence.
Ensure: List N (with n elements) of pairs of processors for which there is a cross-processor A-maximal.

```
 1: for k ← 1 to p do
 2:    V[k] ← k
 3: end for
 4: for k ← 1 to |L| do
 5:    i ← smallest component of L[k]
 6:    j ← largest component of L[k]
 7:    if j > V[i] then
 8:       V[i] ← j
 9:    end if
10: end for
11: n ← 0, k ← 1
12: while k < p do
13:    if V[k] > k then
14:       n ← n + 1
15:       N[n] ← (k, V[k])
16:       k ← V[k]
17:    else
18:       k ← k + 1
19:    end if
20: end while
```

A-maximal. This final scan may be done in time $O(\log(|A|/p))$ if $MList(AP_i)$ is maintained in an array used as a circular buffer and ordered according to the position of the maximals. This is in accordance with Algorithm 1.

So we may claim the following:

Theorem 3. *Using a Coarse Grained Multicomputer with* p *processors, all maximal subsequences of a sequence* A *(already distributed in the* p *local memories) may be found in time* $O(|A|/p)$*, using* $O(|A|/p)$ *local space and* $O(1)$ *communication rounds.*

Proof. Based on the results of this section, along with Lemmas 11, 13, 15 and 16. Notice that only 4 communication rounds are necessary, three of them of size $O(p)$ and one with size $O(|A|/p)$.

4 Conclusion

We have presented an algorithm that finds all maximal subsequences of a sequence A with linear speed-up and high scalability. The size of the communication rounds is bounded by $O(|A|/p)$, but we conjecture that if $|A| \gg p$ the average size of the communication rounds should be much lower than $|A|/p$. In fact, experimenting with a sequence X of random numbers we conjecture that

the average size of $PList(X)$ is $O(\log(|X|))$. The running time of the whole algorithm is dominated by the time of the first step (finding the local maximal subsequences).

It is not trivial to derive this parallel $O(|A|/p)$ time and $O(|A|/p)$ space per processor algorithm, and not intuitive that a parallel algorithm requiring a constant number of communication rounds can be found. We had to explore the properties of those local maximals that are potential candidates to be merged together to form larger maximals, as well as an efficient merge process to join candidate local maximals. This is the reason why we required many auxiliary lemmas.

Some adaptations may be done to this algorithm. For example, it is easy to make it work with a *circular* sequence of numbers, which may be important when dealing with circular chains of nucleotides. Also, if only the best k maximals are needed, a parallel selection algorithm [16] may be used to find the k-th best maximal in $O(|A|/p)$ time and $O(\log p)$ communication rounds.

References

1. C. E. R. Alves, E. N. Cáceres, and S. W. Song. BSP/CGM algorithms for maximum subsequence and maximum subarray. In *Proceedings Euro PVM/MPI 2004 - 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 139–146. Springer Verlag, 2004.
2. J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
3. J. Bentley. *Programming Pearls.* Addison-Wesley, 1986.
4. J. V. Braun and H. G. Müller. Statistical methods for DNA sequence segmentation. *Statist. Sci.*, 13:142–162, 1998.
5. M. Csuros. Algorithms for finding maximal-scoring segment sets. In *Proceedings WABI2004 - 4th Workshop on Algorithms in Bioinformatics*, Lecture Notes in Computer Science. Springer Verlag, 2004.
6. F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings SPAA'95 - ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33. ACM Press, 1995.
7. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307, 1993.
8. L. G. Valiant et al. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–972. MIT Press/Elsevier, 1990.
9. Y. X. Fu and R. N. Curnow. Maximum likelihood estimation of multiple change points. *Biometrika*, 77:563–573, 1990.
10. S. Karlin and V. Brendel. Chance and significance in protein and dna sequence analysis. *Science*, 257:39–49, 1992.
11. R. J. Klein, Z. Misulovin, and S. R. Eddy. Noncoding RNA genes identified in AT-rich hyperthermophiles. *Proc. Natl. Acad. Sci. USA*, 99(11):7542–7547, 2002.
12. W. Li, P. Bernaola-Galván, F. Haghighi, and I. Grosse. Applications of recursive segmentation to the analysis of DNA sequences. *Comput. Chem.*, 26:491–510, 2002.

13. K. Perumalla and N. Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(3):367–373, 1995.
14. K. Qiu and S. G. Akl. Parallel maximum sum algorithms on interconnection networks. Technical report, Queen's Unversity, Department of Computer and Information Science, 1999. No. 99-431.
15. W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241. AAAI Press, August 1999.
16. E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse grained multicomputers. *Algorithmica*, 24:371–380, 1999.
17. L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.
18. Zhaofang Wen. Fast parallel algorithm for the maximum sum problem. *Parallel Computing*, 21:461–466, 1995.