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

Carlos Eduardo Rodrigues Alves[1], Edson Norberto Cáceres[2], and Siang Wun Song[3]

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

Abstract. Given a sequence A of real numbers, we wish to find 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 can be solved sequentially in linear time. We present a BSP/CGM algorithm that uses p processors and takes $O(|A|/p)$ 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.

1 Introduction

Given a sequence of real numbers, the *maximum subsequence problem* finds the contiguous subsequence with the maximum sum [3]. A more general problem is the *all maximal subsequences problem* [8] which finds a list of all non-overlapping contiguous subsequences with maximal sum. These two problems arise in several contexts in Computational Biology. Many applications are presented in [8], to identify transmembrane domains in proteins expressed as a sequence of amino acids and to discover CpG islands. Csuros [4] mentions other applications that require the computation of such subsequences, in the analysis of protein and DNA sequences, determination of isochores in DNA sequences, etc.

Linear time sequential algorithms are known to solve both problems [3, 8]. Wen [10] presents a EREW PRAM algorithm that solves the maximum subsequence problem of n given numbers in $O(\log n)$ time using $O(n/\log n)$ processors.

* Partially supported by CNPq 30.0317/02-6, 30.5218/03-4, 55.0094/05-9 and 62.0123/04-4, and FUNDECT-MS Proc. 41/100117/03. We also acknowlegde the comments of the anonymous referees.

For this same problem, 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. Dai and Su [5] present a PRAM EREW work-optimal algorithm that solve the all maximal subsequences problem in $O(\log n)$ time with $O(n)$ operations.

In this paper we present a BSP/CGM algorithm to solve the all maximal subsequences problem. Given a sequence A of numbers, this algorithm uses p processors and finds all the maximal subsequences in $O(|A|/p)$ time, with $O(|A|/p)$ space per processor, and requires a constant number of communication rounds in which at most $O(|A|/p)$ data are transmitted. Unlike the parallel solution for the basic maximum subsequence problem, it is not at all intuitive that one can find a parallel algorithm for this problem that requires only a constant number of communication rounds. In this sense, this is also an important result in a theoretical viewpoint. Finding a BSP/CGM algorithm with constant number of communication rounds for a problem with linear sequential complexity is not always possible, as shown by the list ranking problem where the best known BSP/CGM algorithm requires $O(\log n)$ communication rounds [7].

2 Preliminary Definitions and Results

Given a sequence A of real numbers, denote its elements by a_i, $1 \leq i \leq |A|$. Subsequences of A are indicated as $A_i^j = (a_{i+1}, ..., a_j)$. The superscript indicates the rightmost position in the subsequence, and the subscript is one less than the leftmost position. If the subscript and the superscript are equal, the subsequence is empty. A particular subsequence of A can be denoted by some other uppercase letter, but 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 notation we have $X = A_{L(X)}^{R(X)} = (a_{L(X)+1}, ..., a_{R(X)})$. 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 $\langle X_1, X_2, ...X_n \rangle$. The sum of the values of a subsequence 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 $Score(A_0^j)$. We consider $PS(0) = 0$. Notice that $Score(A_i^j) = PS(j) - PS(i)$. For a subsequence $X = A_i^j$, the minimum and the maximum among all values of $PS(k)$, for $i \leq k \leq j$, will be denoted by $Min(X)$ and $Max(X)$, respectively.

We consider the BSP/CGM (Bulk Synchronous Parallel/Coarse-Grained Multicomputer) model [9, 6], with p processors each with $O(n/p)$ local memory, where n is the input size of the problem. A BSP/CGM algorithm consists of alternating local computation and global communication rounds. In each communication round, each processor can send/receive messages with at most $O(n/p)$ data. A BSP/CGM algorithm attempts to minimize the number of communication rounds as well as the total local computation time.

A *maximum* scoring subsequence of X is one with the largest score among all scores of subsequences of X. When ties occur, we choose the subsequence of minimum length. If there is no positive number in X, we assume that there is no maximum scoring subsequence. 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 all maximal subsequences of A is more complicated. Ruzzo and Tompa [8] define the set of maximal subsequences recursively, as follows.

Definition 1. *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 the set $\{M\}$, the set of maximal subsequences of A_1, and the set of maximal subsequences of A_2.*

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.

Consider the sequence $A = (a_1, a_2, \ldots, a_{22})$ of 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.

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 [8].

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.*

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.

Definition 2. *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, by the following lemma. In this paper we omit all the proofs. They can be found in [2].

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

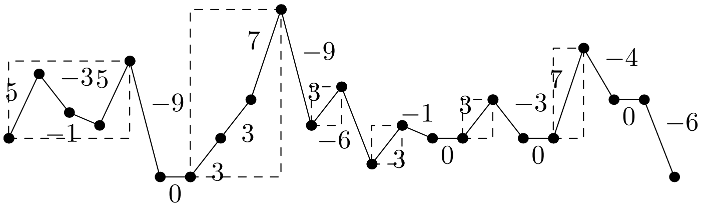

Fig. 2. Sequence $A = (5, -3, -1, \ldots, 0, -6)$ and some Pr1-subsequences.

In Figure 2, we plot the function $PS\,(.)$, so that positive (negative) values in the sequence are represented by ascending (descending) line segments. A Pr1-subsequence X is 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 $PS\,(i) < Min(A_{i+1}^j)$ and it is a *Pr1-suffix* if $Max(A_i^{j-1}) < PS\,(j)$. 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 $Max(P) < Max(S)$.*

One can observe [8] that (i) any Pr1-subsequence of a sequence A is contained in a maximal subsequence of A (maybe not properly), and (ii) given a sequence A, any two distinct maximal subsequences of A do not overlap or touch each other. The parallel algorithm is 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. (As an abuse of our notation we write the plural of X-maximal as X-maximals.) Thus we want to find the set of all A-maximals.

Lemma 2. *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.*

This lemma shows that it is possible to build $MList\,(A)$ working incrementally. Ihis is important for the proposed parallel algorithm, where the 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 $MList\,(X)$, find, if possible, an X-maximal that is a prefix (or suffix) of a larger A-maximal. This clearly involves $MList\,(X)$ and the rest of sequence A. However, some X-maximals need not be considered as possible

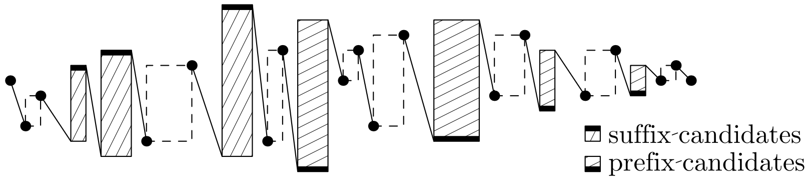

Fig. 3. A sequence X, $\mathit{MList}\left(X\right)$, $\mathit{PList}\left(X\right)$ and $\mathit{SList}\left(X\right)$. The first (last) maximal is not a suffix (prefix) candidate because of the first condition of the definition. The other maximals that are not candidates fall in the second condition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

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. *Given a subsequence X of A, $\mathit{PList}\left(X\right)$ is the ordered list of all X-maximals, with the exception of those X-maximals M for which one of the two conditions are satisfied: (1) $\mathit{Min}(M) \geq \mathit{PS}\left(R\left(X\right)\right)$, 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 $\mathit{PList}\left(X\right)$ are indexed starting at 1 with the leftmost subsequence.)*

Informally, $\mathit{PList}\left(X\right)$ 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 $\mathit{PList}\left(X\right)$ 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 $\mathit{PList}\left(X\right)$ because of condition 1. A_{10}^{11} does not belong to $\mathit{PList}\left(X\right)$ because of both conditions 1 and 2. Thus $\mathit{PList}\left(X\right) = (A_6^9, A_{12}^{13})$.

We have following properties [2]: (i) If X is a subsequence of A, $\mathit{PList}\left(X\right)$ contains all X-maximals that can be a proper prefix of an A-maximal, (ii) if M is a sequence in $\mathit{PList}\left(X\right)$ and $i \in \left]L\left(M\right), R\left(X\right)\right]$ then $\mathit{Min}(M) < \mathit{PS}\left(i\right)$, that is, $A_{L(M)}^{R(X)}$ is a Pr1-prefix, and (iii) if M is a sequence in $\mathit{PList}\left(X\right)$ and $i \in \left]R\left(M\right), R\left(X\right)\right]$ then $\mathit{Max}(M) \geq \mathit{PS}\left(i\right)$. A consequence of these properties is that $\mathit{PList}\left(X\right)$ is in a non-increasing order of $\mathit{Max}(.)$ and a strictly increasing order of $\mathit{Min}(.)$. Figure 3 illustrates $\mathit{PList}\left(X\right)$ (and $\mathit{SList}\left(X\right)$, defined shortly).

We also a need similar definition for possible suffixes of A-maximals. The associated properties are given below. Notice the exchanging roles of $\mathit{Max}(.)$ and $\mathit{Min}(.)$, "left" and "right", etc.

Definition 4. *Given a subsequence X of A, $\mathit{SList}\left(X\right)$ 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) $\mathit{Max}(N) \leq \mathit{PS}\left(L\left(X\right)\right)$, or (2) there is a X-maximal M to the left of N such that $\mathit{Max}(N) \leq \mathit{Max}(M)$. (The elements of $\mathit{SList}\left(X\right)$ are indexed starting at 1 with the rightmost subsequence.)*

Properties: (i-a) If X is a subsequence of A, $\mathit{SList}\,(X)$ contains all X-maximals that can be a proper suffix of an A-maximal, (ii-a) if N is a sequence in $\mathit{SList}\,(X)$ and $i \in [L\,(X)\,, R\,(N)[$ then $\mathit{PS}\,(i) < \mathit{Max}(N)$, that is, $A_{L(X)}^{R(N)}$ is a Pr1-suffix, and (iii-a) if N is a sequence in $\mathit{SList}\,(X)$ and $i \in [L\,(X)\,, L\,(N)[$ then $\mathit{PS}\,(i) \geq \mathit{Min}(N)$. Notice that at most one X-maximal may belong to both $\mathit{PList}\,(X)$ and $\mathit{SList}\,(X)$, namely a maximum subsequence of X. Any other element of $\mathit{SList}\,(X)$ must be to the left of any element of $\mathit{PList}\,(X)$. See Figure 3 for an illustration of $\mathit{PList}\,(X)$ and $\mathit{SList}\,(X)$ (when these lists are disjoint).

3 The Parallel Algorithm

Consider p processors $P_1, P_2, \ldots, P_p$. Assume that the input sequence 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, each 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. Lemma 3 shows how to find the local maximals.

Lemma 3. *In $O(|A|/p)$ time and space and with one communication round of size $O(p)$, each processor P_i may obtain: (i) its local lists of maximals ($\mathit{MList}\,(AP_i)$), prefix candidates ($\mathit{PList}\,(AP_i)$) and suffix candidates ($\mathit{SList}\,(AP_i)$), and (ii) $\mathit{PS}\,(L\,(AP_j))$, $\mathit{Min}(AP_j)$ and $\mathit{Max}(AP_j)$ for* all $j \in [1, p]$.

We now consider a basic procedure to join lists of maximals. We will see how $\mathit{MList}\,(Z)$ may be obtained from $\mathit{MList}\,(X)$, $\mathit{MList}\,(Y)$, $\mathit{PList}\,(X)$ and $\mathit{SList}\,(Y)$ when $Z = \langle X, Y \rangle$. The following lemma states the condition for two local maximal subsequences to be merged to form a larger one.

Lemma 4. *Given $M \in \mathit{PList}\,(X)$ and $N \in \mathit{SList}\,(Y)$, $A_{L(M)}^{R(N)}$ is a Pr1-subsequence iff $\mathit{Min}(M) < \mathit{Min}(N)$ and $\mathit{Max}(M) < \mathit{Max}(N)$.*

Properties (i) and (i-a) state that we may search for a Z-maximal that overlaps X and Y using only $\mathit{PList}\,(X)$ and $\mathit{SList}\,(Y)$. Algorithm 1 does this. We use $\mathit{Pl} = \mathit{PList}\,(X)$ and $\mathit{Sl} = \mathit{SList}\,(Y)$ for short, indexing them as stated in Definitions 3 and 4. The algorithm returns the indices of the chosen candidates for prefixes and suffixes of the new Z-maximal.

Algorithm 1: 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.
$i_p \leftarrow 1$, $i_s \leftarrow 1$, $f \leftarrow false$
while $i_p \leq |Pl|$ and $i_s \leq |Sl|$ and not $f$ do
    if $Max(Pl[i_p]) \geq Max(Sl[i_s])$ then
        $i_p \leftarrow i_p + 1$
    else if $Min(Pl[i_p]) \geq Min(Sl[i_s])$ then
```

```
        $i_s \leftarrow i_s + 1$
    **else**
        $f \leftarrow true$
    **end if**
**end while**
```

It can be shown that, given $Z = \langle X, Y \rangle$, $Pl = PList\,(X)$ and $Sl = SList\,(Y)$, Algorithm 1 finds the only Z-maximal that overlaps X and Y, if it exists, in $O(|Pl| + |Sl|)$ time and $O(1)$ additional space.

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 all the A-maximals that cross processor boundaries must be contained in this set.

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 5. *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 $Max(N) > \max_{i<k<j} Max(AP_k)$.*

After the local step described in Lemma 3 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 of Lemma 5 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.

Lemma 6. *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 $O(p)$.*

Algorithm 2 presents the tagging process, based on a case by case study [2]. This algorithm contains the tagging procedure for the prefix candidates of $PList\,(AP_i)$, called Pl for short. $PTagList\,(i)$ is called Tl for short. Figure 4 illustrates the tagging of prefix candidates.

Algorithm 2: Tagging a List of Prefix Candidates

Require: Lists Pl and Tl, with $|Pl|$ and $|Tl|$ elements, respectively.

Ensure: Tagging of the elements of Pl.

```
$i_p \leftarrow 1$, $i_t \leftarrow 1$, $f \leftarrow false$
```

```
while $i_p \leq |Pl|$ and $i_t \leq |Tl|$ and not $f$ do
  if $Max(Pl[i_p]) \geq Max(Tl[i_t])$ then
    $i_p \leftarrow i_p + 1$
  else if $Min(Pl[i_p]) \geq Min{*}(Tl[i_t])$ then
    $i_t \leftarrow i_t + 1$
  else
    tag $Pl[i_p]$ with $tag(Tl[i_t])$
    if $Min(Pl[i_p]) < Min(Tl[i_t])$ then
      $f \leftarrow true$
    end if
  end if
end while
```

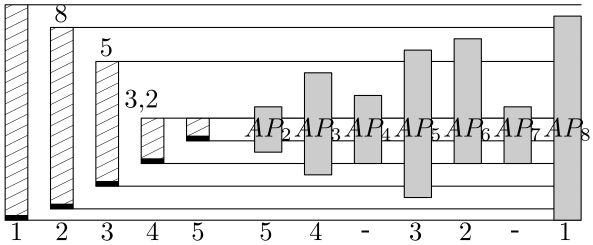

Fig. 4. We consider the tagging of elements of *PList* (AP_1), represented as shaded bars on the left. The darkened bars in the right represent the data from other processors. The numbers below the bars represent the indices in *PList* (AP_i) and *PTagList* (1).

After the tagging procedure, 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 *PList* (AP_i) that are tagged with j and elements of *SList* (AP_j) that are tagged with i. These elements must be in the local memory of one single processor for verification by Algorithm 1. 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.

The following lemma summarizes the complexity of the tagging process.

Lemma 7. *After tagging the prefix and suffix candidates, 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 to find the Pr1-subsequences that are really A-maximals. All processors broadcast the information about the new Pr1-subsequences found. Every processor then eliminates the Pr1-subsequence that are contained in another Pr1-subsequence. The presented procedure does not generate two Pr1-subsequences that overlap, unless one is contained in the other. That is because if two Pr1-subsequences overlap then their union is also a Pr1-subsequence. Each Pr1-subsequence is related to a different pair of processors. All that must be verified is which pairs generated new sequences, done by Algorithm 3. It takes $O(p)$ time and space.

Algorithm 3: Removing Pr1-subsequences that 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.

```
for $k \leftarrow 1$ to $p$ do
    $V[k] \leftarrow k$
end for
for $k \leftarrow 1$ to $|L|$ do
    $i \leftarrow$ smallest component of $L[k]$
    $j \leftarrow$ largest component of $L[k]$
    if $j > V[i]$ then
        $V[i] \leftarrow j$
    end if
end for
$n \leftarrow 0$, $k \leftarrow 1$
while $k < p$ do
    if $V[k] > k$ then
        $n \leftarrow n + 1$
        $N[n] \leftarrow (k, V[k])$
        $k \leftarrow V[k]$
    else
        $k \leftarrow k + 1$
    end if
end while
```

A final step is done locally by each processor. By examining the list of new A-maximals, processor P_i verifies if there is an A-maximal that contains its entire local subsequence AP_i, which means that its own local set of maximals $\mathit{MList}\,(AP_i)$ should be discarded. This 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 $\mathit{MList}\,(AP_i)$ will eliminate the local maximals that are contained in a larger A-maximal. This final scan can be done in time $O\left(\log(|A|/p)\right)$.

Theorem 2. *Using a BSP/CGM with p processors, all maximal subsequences of a sequence A (already distributed in the p local memories) can be found in time $O(|A|/p)$, using $O(|A|/p)$ local space and O(1) communication rounds.*

4 Conclusion

We have presented a parallel 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 $O(|A|/p)$. communication rounds should be much lower than $|A|/p$. 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 to find the local maximal subsequences. To derive this parallel $O(|A|/p)$ time and $O(|A|/p)$ space per processor algorithm, requiring a constant number of communication rounds, we explored 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.

References

1. C. E. R. Alves, E. N. Cáceres, and S. W. Song. BSP/CGM algorithms for maximum subsequence and maximum subarray. In *Proceedings 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 139–146. Springer Verlag, 2004.
2. C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for finding all maximal contiguous subsequences of a sequence of numbers. Technical report, Universidade de São Paulo, January 2005.
3. J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.
4. M. Csuros. Algorithms for finding maximal-scoring segment sets. In *Proceedings WABI2004 - 4th Workshop on Algorithms in Bioinformatics*, volume 3240 of *Lecture Notes in Computer Science*, pages 62–73. Springer Verlag, 2004.
5. H.-K. Dai and H.-C. Su. A parallel algorithm for finding all successive minimal maximum subsequences. In *Latin American Theoretica Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 337–348. Springer Verlag, 2006.
6. 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.
7. F. Dehne, A. Ferreira, E. Caceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.
8. W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology*, pages 234–241. AAAI Press, August 1999.
9. L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.
10. Z. Wen. Fast parallel algorithm for the maximum sum problem. *Parallel Computing*, 21:461–466, 1995.