Algorithmica (2009) 53: 35–49
DOI 10.1007/s00453-007-9006-9

Algorithms for Maximum Independent Set in Convex Bipartite Graphs

José Soares · Marco A. Stefanes

Received: 27 June 2007 / Accepted: 29 June 2007 / Published online: 22 September 2007

Abstract A bipartite graph $G = (V, W, E)$ is *convex* if there exists an ordering of the vertices of W such that, for each $v \in V$, the neighbors of v are consecutive in W. We describe both a sequential and a BSP/CGM algorithm to find a maximum independent set in a convex bipartite graph. The sequential algorithm improves over the running time of the previously known algorithm and the BSP/CGM algorithm is a parallel version of the sequential one. The complexity of the algorithms does not depend on $|W|$.

Keywords Convex bipartite graphs · Independent sets · BSP/CGM algorithms · Parallel algorithm

1 Introduction

Bipartite convex graphs were introduced by Glover [9], motivated by some industrial applications. Since then several algorithms have been developed for problems in this kind of graph [2, 3, 8, 12, 13].

Let $G = (V, W, E)$ be a bipartite graph, where V and W define the bipartition of the vertices, and E is the edge set in the form (v, w), where $v \in V$ and $w \in W$. The graph G is *convex* if the vertices in W can be ordered in such a way that, for each

---

This work was supported by FAPESP (Proc. 98/06327-0). The first author was also supported by FAPESP (Proc. 96/04505–2), and CNPq/MCT/FINEP (PRONEX project 107/97).

J. Soares (✉)
Instituto de Matemática e Estatística, Universidade de São Paulo, Rua do Matão 1010,
05508-900 São Paulo, SP, Brazil
e-mail: jose@ime.usp.br

M.A. Stefanes
Departamento de Computação e Estatística, Universidade Federal de Mato Grosso do Sul,
Campo Grande, Brazil
e-mail: marco@dct.ufms.br

$v \in V$, the neighbors of v are consecutive in W. For convenience, we consider that $V = \{1, \dots, |V|\}$, $W = \{1, \dots, |W|\}$, and that the vertices in W are given according to the ordering mentioned above. This ordering can be obtained in a preprocessing step by a linear time sequential algorithm [1], or by a BSP/CGM algorithm with linear time per round and $O(\log^2 p)$ communication rounds [3].

As an application of the use of bipartite convex graphs, we mention the following problem, which is a simplification of a situation reported by Glover [9]. The problem is to assembly left halves from a set V with right halves from a set W for manufacturing a certain product. Each half h in $V \cup W$ has a size $s(h)$. Assume that $v \in V$ can be assembled with $w \in W$ only if $L \leq s(v) - s(w) \leq U$, where U and L are given constants. Then, the maximum number of halves that can be matched is equal to the size of a *maximum matching* (see below) in the convex bipartite graph $G = (V, W, E)$ where $E = \{(v, w) \mid L \leq s(v) - s(w) \leq U\}$. Other applications are mentioned by Lipski and Preparata [12] and by Dekel and Sahni [8].

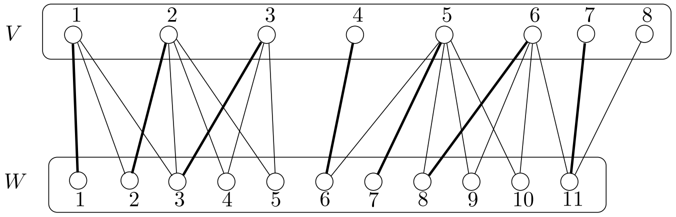
We say that a vertex $w' \in W$ is smaller (larger) than a vertex $w'' \in W$ if the integer representing w' is smaller (larger) than the integer representing w''. A convex bipartite graph has a *compact* representation by a set of $|V|$ triples of the form $(i, \mathit{begin}(i), \mathit{end}(i))$, where i is a vertex in V, $\mathit{begin}(i)$ and $\mathit{end}(i)$ are the smallest and largest vertices, respectively, in the interval of vertices of W connected to i.

A *matching* M in a graph G is a subset of the edges such that no two edges in M has a common endpoint vertex. A matching is *maximum* if its cardinality is as large as possible. A vertex x is *matched by* M if there is an edge in M incident to x. If a vertex x is not matched by M, we say that x is *free* with respect to M. A matching M in a bipartite convex graph is *greedy* if it has the following properties:

1. if $(i, j) \in M$, then, for each $j' \in W$, with $\mathit{begin}(i) \leq j' \leq j - 1$, there exists $i' \in V$ such that $(i', j') \in M$ and $\mathit{end}(i') \leq \mathit{end}(i)$;
2. if $j \in W$ is adjacent to a free vertex $i' \in V$, then there exists $i \in V$, with $\mathit{end}(i) \leq \mathit{end}(i')$, such that $(i, j) \in M$.

Figure 1 shows a convex bipartite graph in its compact representation and a greedy matching.

A greedy matching can be obtained by visiting the elements of W in ascending order: for each free $j \in W$, find a vertex i with the smallest $\mathit{end}(i)$ among the free vertices of V adjacent to j, and add the edge (i, j) to M. This algorithm runs in



V	$begin$	end
1	1	3
2	2	5
3	3	5
4	5	6
5	6	10
6	8	11
7	11	11
8	11	11

Fig. 1 Compact representation and a greedy maximum matching (bold edges)

$O(|E|)$ time and is known as Glover's algorithm [9]. The matching obtained in this fashion is indeed maximum.

We now comment on some previous algorithms for convex bipartite graphs. Let $n := |V|$, $m := |W|$, $N := |V| + |W|$, and p be the number of processors used by the parallel algorithm.

Steiner and Yeomans [13] designed an $O(n)$ sequential algorithm to compute a maximum matching in a convex bipartite graph. Dekel and Sahni [8] developed an EREW PRAM algorithm for this problem which runs in time $O(\log^2 n)$ and requires $O(n)$ processors. Bose et al. described [2] a CGM algorithm that requires $O(\log p)$ communication rounds and $O(T_s(n/p, m/p) + (n/p)\log p)$ local computation time, where $T_s(x, y)$ is the sequential complexity for the same problem with $|V| = x,\ |W| = y$. Bose et al. also described a BSP algorithm that requires $O(\log p)$ supersteps with $O(T_s(n/p, m/p) + (n/p)\log p)$ local computation time, and $O(gN + (gn/p)\log p)$ communication cost. All the matchings obtained by these algorithms are greedy.

An *independent set* in a graph is a set of vertices such that no edge connects two vertices in the set. A *maximum independent set* (MIS) is an independent set which has maximum cardinality. Lipski and Preparata [12] presented a sequential $O(N)$ algorithm that receives a compact representation of a convex bipartite graph G, and a greedy matching M of G and returns a MIS. Czumaj et al. [6] described a CRCW PRAM algorithm that runs in $O(\log N)$ time with $O(N/\log N)$ processors.

In this work, we present both a sequential and a BSP/CGM parallel algorithm to compute a MIS in a convex bipartite graph. The input to our algorithms is a convex bipartite graph G and a greedy matching of G. The sequential algorithm is linear in n provided that $m = O(n^c)$, for some constant c. This algorithm improves on the worst case complexity for the problem. Using p processors, the BSP/CGM algorithm requires a constant number of communication rounds in which each processor sends and receives messages of size $O(n/p)$, and $O(n/p)$ local computation time, assuming that $n \geq p^2$. This implies a BSP algorithm with constant number of supersteps, $O(n/p)$ computation time and $O(gn/p)$ communication cost.

We first, in Sect. 2, give a review of the parallel computation models. Next, in Sect. 3, we present the sequential algorithm to compute a MIS in a convex bipartite graph and its analysis. In Sect. 4, we describe the BSP/CGM parallel algorithm, which is based on the sequential one, and we address its time complexity. Finally, in Sect. 5, some concluding remarks are given.

2 The Models BSP and CGM

The model BSP [14] (*Bulk Synchronous Parallel*) was one of the first models of parallel computation that takes into account the communication costs. This simple model has showed success in predicting the practical behavior of algorithms.

A BSP algorithm consists in a sequence of *supersteps* with a synchronization barrier at the end of each superstep. In one superstep the processors operate independently performing local computations and global communications. We say that a

h-relation is performed in a superstep when each processor sends or receives at most h messages. We consider that each message has a fixed size, depending only on the parallel machine. A message sent will be available in its destination for processing by the next superstep.

In the BSP model the processors communicate through some arbitrary interconnection network provided with a facility for synchronization. The model has three parameters: the number p of processors; the minimum time L of a superstep; and the quotient g between the number of local operations per second performed by all the processors and the total number of messages that can delivered per second. The time of a superstep in which a h-relation is performed is $\max\{gh + w, L\}$, where w is the time spent in local computations during the superstep. The running time of a BSP algorithm is the sum of the time of its supersteps.

The model CGM [7] (*Coarse Grained Multicomputer*) is a version of the model BSP consisting of p processors, where each one has $O(n/p)$ local memory. It is usual to assume that $n/p \geq p^{\epsilon}$, for some fixed $\epsilon > 0$. As in the BSP model, processors can communicate through some arbitrary interconnection network. A CGM algorithm consists of local computation alternated with global communication. In a communication round, each processor can send or receive $O(n/p)$ values. The running time of a CGM algorithm is the sum of the time of each round of computation and communication.

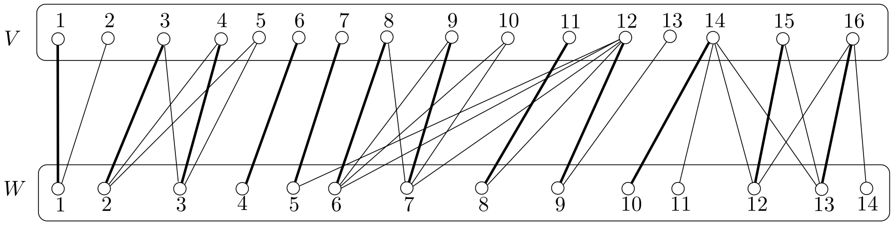
3 Sequential Algorithm

Let $G = (V, W, E)$ be a convex bipartite graph and M be a matching in G. We call a path in G *alternating* with respect to M if the path starts at a free vertex of V (with respect to M) and whose edges are, alternatively, in M and in $E \setminus M$. Thus, if e and e' are consecutive edges in an alternating path, then $e \in E$ and $e' \in E \setminus M$, or vice-versa. A vertex v is *reachable* if there exists an alternating path ending at v. Note that, by this definition, an alternating path always has its beginning in a free vertex of V.

It is well-known that a maximum independent set in a bipartite graph can be derived from a maximum matching using standard alternating path techniques [11]. If M is a maximum matching in G and letting $V_R \subseteq V$ and $W_R \subseteq W$ be the set of reachable vertices, then $I = V_R \cup (W \setminus W_R)$ is a maximum independent set. So, the entire problem reduces to find the reachable vertices.

We denote by $[i, j]$ the set of integers $\{i, i + 1, \dots, j\}$. Thus, $V = [1, n]$ and $W = [1, m]$. Abusing notation, we let V also denote an array representing G in a compact representation, along with a greedy matching M. Each element of the array $V[1..n]$ has the fields *begin*, *end*, and M. The triple $(i, \mathit{begin}(i), \mathit{end}(i))$ of the compact representation of G is represented here by $(i, V[i].\mathit{begin}, V[i].\mathit{end})$. The field M represents a matching in G. For each $i \in V$, $V[i].M = j > 0$ if $(i, j) \in M$, and $V[i].M = 0$ if i is a free vertex.

The input to the algorithm is a convex bipartite graph G and a greedy matching of G. We may assume that the graph has no isolated vertices since isolated vertices are always in a MIS. The algorithm described below begins by adding a new field,



V	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
label	1	1	2	3	3	4	5	6	7	7	8	9	9	10	12	13

Fig. 2 Convex bipartite graph, greedy matching (bold edges), and labels of V

label, to the array V. Each vertex $i \in V$ has label $V[i].M$ if i is a matched vertex, or label $V[i].end$ if i is a free vertex. Thus, if two vertices have the same label, at least one of them is free. The algorithm sorts V according to these labels. Ties are broken in such a way that matched vertices come first. Figure 2 shows a convex bipartite graph, a greedy maximum matching and the corresponding labels of the vertices of V.

The fact below was observed by Czumaj, Diks, and Przytycka [6].

Fact 3.1 If $i \in V$ is a free vertex with respect to a greedy matching M, then the vertices of W reachable by alternating paths beginning at i forms an interval in $[1, V[i].end]$.

A similar key fact is exploited by our algorithms: the ordering by labels of V guarantees that vertices of V reachable by alternating paths beginning at a free vertex of V form an interval in $[1, n]$. The criterion used to break ties in the ordering by labels of V is needed to ensure that the intervals of V are computed correctly.

In the algorithm, the arrays V_R and W_R represent the set of vertices reachable by alternating paths. For each i, the intervals $[V_R[i].begin, V_R[i].end] \subseteq [1, n]$ and $[W_R[i].begin, W_R[i].end] \subseteq [1, m]$ correspond to reachable vertices in V and in W. Again, abusing notation, we will also consider V_R and W_R as the union set of the intervals represented by the arrays V_R and W_R.

After the sorting, the **Procedure MIS** is called with the array $V[a..b]$ as input, where $a = 1$ and $b = n$. The array V is inspected beginning in position b. The algorithm searches for a free vertex i. When such a vertex is reached, the values of *begin_V* and *begin_W* are initialized with i and $V[i].begin$, respectively. Vertices in V are inspected in decreasing order of labels, until the largest i', $a \le i' \le i$ is reached, such that: (1) $V[i'].label = begin\_W$; (2) $i' - 1 < a$ or $V[i' - 1].label < begin\_W$; (3) for each i'', $begin\_V \le i'' \le end\_V$, $V[i''].begin \ge begin\_W$. To satisfy these conditions, the algorithm alters the values of *begin_V* and *begin_W* when necessary.

Once the arrays V_R and W_R are obtained, the **Procedure Union** builds a representation of the corresponding maximum independent set $I = V_R \cup (W \setminus W_R)$. The output of the **Procedure Union** consists of arrays I_V and I_W. The array I_V represents the same intervals of vertices represented by V_R, while the array I_W represents the intervals of vertices in $W \setminus W_R$. By construction, the intervals represented in V_R are such that $V_R[i+1].end < V_R[i].begin$, for $1 \leq i < j$, where j is the number of intervals. The same holds for the intervals represented in W_R. So, arrays I_V and I_W built by the **Procedure Union** are such that

$$I_V = \bigcup_{i=j}^{1} [V_R[i].begin, V_R[i].end],$$

and

$$\begin{aligned} I_W = {} & [1, W_R[j].begin - 1] \cup \bigcup_{i=j-1}^{1} [W_R[i+1].end + 1, W_R[i].begin - 1] \\ & \cup [W_R[1].end + 1, m]. \end{aligned}$$

While the number of intervals in I_V is j, the number of intervals in I_W is $j+1$. Observe that the representation of I by intervals is necessary to keep the complexity of our algorithm independent of m, since a maximum independent set may have size $\Omega(m)$.

Sequential Algorithm

Input: A convex bipartite graph $G = (V, W, E)$ without isolated vertices and a greedy matching of G, given in the array $V[1..n]$.

Output: Arrays I_V and I_W representing intervals of vertices of a maximum independent set.

```
Create a new field in the array $V$, the field label.
for $i := 1$ to $n$ do
    if $V[i].M > 0$ then $V[i].label := V[i].M$
    else $V[i].label := V[i].end$
Sort the array $V$ in nondecreasing order of labels. Ties are broken in such a way that
    matched vertices come first.
Call Procedure MIS with input $V[1..n]$ to obtain arrays $V_R$, $W_R$, and integer $j$
Call Procedure Union with input $V_R$, $W_R$, $j$, 1, and $m$ to obtain independent sets $I_V$
    and $I_W$.
Return $I_V$ and $I_W$.
```

Procedure MIS

Input: An array $V[a..b]$ with fields $V[i].begin$, $V[i].end$, $V[i].M$ and $V[i].label$. The array is ordered by the field $V[i].label$ with the ties broken by putting the vertex that participates in a matching first.

Output: Arrays V_R and W_R of size j representing vertices reachable by alternating paths that originate in free vertices of $V[a..b]$.

```
1: $i := b$, $j := 0$
2: **while** $i > a$ **do**
3:   **if** $V[i].M = 0$ **then** {a free vertex is found}
4:     $begin\_V := i$
5:     $end\_V := i$
6:     $begin\_W := V[i].begin$
7:     $end\_W := V[i].label$
8:   **repeat**
9:     $begin\_V := i$ {Vertex $i$ is inserted in $[begin\_V, end\_V]$}
10:    $begin\_W := \min\{V[i].begin, begin\_W\}$
11:    **Invariant 1** Each vertex in $[begin\_V, end\_V] \subseteq V$ is reachable from some free vertex in $V$ by an alternating path
12:    **Invariant 2** Each vertex in $[begin\_W, end\_W] \subseteq W$ is reachable from some free vertex in $V$ by an alternating path
13:    $i := i - 1$
14:  **until** $i < a$ **or** $V[i].label < begin\_W$
15:  $j := j + 1$
16:  $V_R[j].begin := begin\_V$
17:  $V_R[j].end := end\_V$
18:  $W_R[j].begin := begin\_W$
19:  $W_R[j].end := end\_W$
20:  **Invariant 3** Each vertex in $[begin\_V, end\_V]$ has its label in $[begin\_W, end\_W]$
21:  **else**
22:    $i := i - 1$
23: Return $V_R$, $W_R$, and $j$
```

Procedure Union

Input: Arrays V_R and W_R of size j representing vertices reachable by alternating paths that originate in free vertices of V, and c and d, two integers representing the interval $[c, d]$ of W to be considered.

Output: Arrays I_V and I_W representing intervals of vertices of a maximum independent set.

```
$I_W[1].begin := c$
**for** $i := 1$ **to** $j$ **do**
  $I_V[i].begin := V_R[j - i + 1].begin$
  $I_V[i].end := V_R[j - i + 1].end$
  $I_W[i].end := W_R[j - i + 1].begin - 1$
  $I_W[i + 1].begin := W_R[j - i + 1].end + 1$
$I_W[j + 1].end := d$
Return $I_V$ and $I_W$
```

The following propositions show the correctness of the algorithm.

Lemma 3.2 *Invariants* 1 *and* 2 *of the* ***Sequential Algorithm*** *are correct.*

Proof Let i be a value for which the condition in Line 3 is true. We will show, by induction on the number of times that the command **repeat** is executed, that the invariants are true.

The first time that the command **repeat** is executed, we have that $[begin\_V, end\_V] = [i, i]$ and $[begin\_W, end\_W] = [V[i].begin, V[i].end]$. Since i is free, and therefore reachable by the definition of alternating paths, also the vertices in $[V[i].begin, V[i].end]$ are reachable.

Consider now an arbitrary iteration of the command **repeat**, and suppose that in the last iteration the invariants were true. Consider the values of i and $begin\_W$ in the beginning of this iteration. By induction, we know that the vertices in $[i+1, end\_V]$ are reachable. In order to insert the vertex i in the interval $[begin\_V, end\_V]$, the value of $begin\_V$ is changed. Note that if $V[i].label \geq begin\_W$, then $(i, V[i].label) \in M$ or i is free. In both cases we have that i is also reachable. Since we are in a new iteration, the condition to leave the loop **repeat** is false and it holds that $V[i].label \geq begin\_W$. Thus, $V[i].label \in [begin\_W, end\_W]$ (remember that the array V is ordered by labels and that the value of i always decreases during the execution of the algorithm).

Since, by induction, each vertex in $[begin\_W, end\_W]$ is reachable, the vertex i is also reachable and it follows that each vertex in $[i = begin\_V, end\_V]$ is reachable. From i being reachable, it follows that all vertices in $[V[i].begin, V[i].end]$ are also reachable. Furthermore, since we know that $V[i].label \in [begin\_W, end\_W]$, each vertex of W in $[begin\_W = \min\{V[i].begin, begin\_W\}, end\_W]$ is reachable. ☐

Lemma 3.3 The Invariant 3 of the Sequential Algorithm is correct.

Proof When the condition of the command **if** in Line 3 is true, the command **repeat** is executed by the first time with initial value of end_W equal to $V[end\_V].label$. The value of end_W is not changed during execution of **repeat**. When the command **repeat** finishes, we have that $V[begin\_V].label \geq begin\_W$.

During execution of the command **repeat**, values of i are considered in decreasing order. Since the array V is sorted by labels, the above observations imply that $begin\_W \leq V[begin\_V].label \leq V[end\_V].label = end\_W$ at the end of the **repeat**, proving the lemma. ☐

In what follows, I is the set $I = V_R \cup W \setminus W_R$.

Lemma 3.4 *Let $V_F \subseteq V$ and $W_F \subseteq W$ be the set of free vertices of G with respect to M. Then, $|I| \geq |M| + |V_F| + |W_F|$.*

Proof Note that $V_F \subset V_R$, since free vertices are considered and inserted in V_R either in Line 3 or during execution of command **repeat**.

It is also true that $W_F \subset W \setminus W_R$, because, by Invariant 2, every vertex in W_R is reachable by alternating paths, and, therefore, it cannot be free: an alternating path ending in a free vertex would indicate the existence of a matching with cardinality larger than the cardinality of the maximum matching M.

We will argue now that each edge in M is incident with at least a vertex in I. Suppose that $(i, j) \in M$ and $j \notin I$. Then, by the definition of I, $j \in W_R$ and, by consequence, there exists k such that $j \in [W_R[k].begin, W_R[k].end]$. By Lemma 3.3, the vertex i, whose label is j, is in the interval $[V_R[k].begin, V_R[k].end]$, and, therefore, is in $V_R \subseteq I$.

Thus, the lemma is true since the following three sets are pairwise disjoints: sets of matched vertices, the set V_F, and the set W_F. ☐

Lemma 3.5 The set I is independent.

Proof We will show that, for each k, the vertices in W which are neighbors to vertices in $[V_R[k].begin, V_R[k].end]$ are all in the interval $[W_R[k].begin, W_R[k].end]$. Since $I = V_R \cup W \setminus W_R$, the lemma follows.

Suppose that for some k, there exists $i \in [V_R[k].begin, V_R[k].end]$ with some neighbor not in $[W_R[k].begin, W_R[k].end]$. Then, either $V[i].begin < W_R[k].begin$, or $V[i].end > W_R[k].end$.

The first case cannot happen. When i was inserted in the interval $[begin\_V, end\_V]$ (Line 9), it is checked in the command **if** whether $V[i].begin < begin\_W$ and, if it is the case, the value of $begin\_W$ is changed. Since the value of $begin\_W$ never increases in the loop **repeat**, at the end of one of its execution the value of $V[i].begin$ continues less or equal to $begin\_W$.

Let us inspect the second case, when $V[i].end > W_R[k].end$. If it is true, since i is reachable, there would exist vertices in W reachable from a free vertex q, with larger values than $V[q].label = W_R[k].end$. But, by Fact 3.1, this also cannot occur. □

Lemma 3.6 *Let $G = (V, W, E)$ be a bipartite graph, I an independent set in G and M a matching in G. Then, $|I| \leq |V_F| + |W_F| + |M|$, where $V_F \subseteq V$ and $W_F \subseteq W$ are the set of free vertices of G with respect to M.*

Proof Note that $|V| + |W| = |V_F| + |W_F| + 2|M|$, since each vertex in G is either free or matched. It follows that if $|I| > |V_F| + |W_F| + |M|$, then I contains more than M matched vertices. Therefore, there exists at least one edge in M connecting two vertices of I, contradicting the definition of independent set. □

Finally, we have the theorem that finishes the correctness of the algorithm.

Theorem 3.7 *Let $G = (V, W, E)$ be a bipartite convex graph without isolated vertices. Then, the set $I = V_R \cup W \setminus W_R$ is a maximum independent set, where V_R and W_R are the sets determined by the Sequential Algorithm.*

Proof It follows directly from Lemmas 3.5, 3.4, and 3.6 that I is a maximum independent set. □

To finish this section, we comment on the time complexity of the **Sequential Algorithm**. The initialization of the field of V containing the labels can be clearly done in time $O(n)$. The ordering of array V can be done in time $O(n)$ using Radixsort [5, Chap. 9], provided that $m = O(n^c)$ for some constant c. Otherwise, we can use a standard $O(n \log n)$ sorting algorithm. To verify that the command **while** of **Procedure MIS** can be done in time $O(n)$, it is enough to note that the value of i, initially n, always decreases of at least one in each iteration of the loop **while** or in each iteration of the loop **repeat**. **Procedure Union** takes time $O(j)$. Since $j \leq n$, the procedure takes time $O(n)$. Therefore, the **Sequential Algorithm** runs in time $O(n) + T_s(n, m)$, where $T_s(x, y)$ is the time to sort x integers belonging to the interval $[1, y]$.

Another sequential algorithm, due to Lipski and Preparata [12], is known for this problem. Their algorithm runs in time $\Theta(n+m)$. Note that our algorithm improves in the worst case complexity for the problem. When $m=\Theta(n^c)$ for some $c>1$, the Lipski and Preparata algorithm runs in time $\Theta(n^c)$, while ours is linear in n. Otherwise, say, $m=\Omega(n^c)$ for some constant $c>1$, the Lipski and Preparata algorithm runs in time $\Omega(n^c)$, while ours runs in time $O(n\log n)$.

4 BSP/CGM Algorithm

The BSP/CGM algorithm is a parallel version of the sequential algorithm. The input to the algorithm is the array $V[1..n]$, which is equally distributed among the available processors. Likewise in the sequential algorithm, the vertices of V are labeled and sorted by labels. Then, V is redistributed to the processors. As we shall see, the labeling can be done in linear time without communication. The sorting can be yielded using Chan and Dehne's algorithm [4] or Goodrich's algorithm [10].

Assume that there are p available processors $P_1, P_2, \dots, P_p$. Let $V[a..b]$ be the input to processor P_k, $1\le k\le p$ after the ordering of the vertices according to their labels. We say that the vertices in $V[a..b]\cup[V[a].\mathit{label}, V[b].\mathit{label}]$ are *attributed* to processor P_k. Although it is possible that some vertices of W are not attributed to any processor, it is true that every reachable vertex in W is attributed to some processor. As output, each P_k will determine intervals of vertices belonging to a maximum independent set in the graph.

In the sequential algorithm, the vertices in V are visited in descending order. From the correctness of **Procedure MIS**, there cannot be the case that alternating paths beginning in free vertices attributed to P_q, with $q<k$, reach vertices attributed to P_k. So, the problem to be solved in the parallel context is how processor P_k detects the existence of alternating paths with origin in vertices attributed to processors $P_{k+1}, P_{k+2}, \dots, P_p$, reaching vertices attributed to P_k. And, in the positive case, how to know which vertices are reachable by these paths. To do that, it is enough to know the value of *min_reach*, where min_reach is the least vertex of W reachable by an alternating path beginning in a free vertex attributed to P_q, with $q>k$. Once min_reach is known, processor P_k can proceed its processing as the **Sequential Algorithm**. The problem now is how to determine the value of *min_reach* without depending on the chained result of processors $P_p, \dots, P_{k+2}, P_{k+1}$.

Suppose that alternating paths beginning in free vertices attributed to processor P_{k+1} reach vertices attributed to P_k. Let *min_rel* be the vertex of W with the smallest number that is reachable by these alternating paths leaving vertices of V attributed to P_{k+1}. The computation of *min_rel* is done locally in parallel and it is communicated to all processors. Then, processor P_k constructs an array $\mathit{Min\_rel}[k+1..p]$ containing the number of these vertices. The minimum of the values in the array is a candidate to be *min_reach*.

Although this information is necessary, it is not sufficient. It might be the case that there exist alternating paths beginning in free vertices attributed, for instance, to P_{k+2}, that reach vertices attributed to P_k. If such paths do not use vertices attributed to P_{k+1}, they can be detected by processor P_k consulting the value of $Min\_rel[k+2]$. Otherwise, all vertices attributed to P_{k+1} are reachable. In this case, the value of *min_reach* will be the minimum between $Min\_rel[k+2]$ and the minimum among all $V[i].begin$, for all $i \in V$ attributed to P_{k+1}. This last minimum, called *min_abs*, can be also computed in parallel and communicated to all processors. Then, processor P_k constructs an array $Min\_abs[k+1..p]$ containing the numbers of these vertices.

However, we have yet another problem. It can be the case that all alternating paths that originate in vertices attributed to processor P_{k+2} finish in vertices attributed to processor P_{k+1}, not reaching vertices attributed to P_k. To detect this situation, processor P_{k+1} communicates whether there is a vertex attributed to it that is a candidate to be the endpoint of an interval of reachable vertices. Lemma 3.1 states that vertices that are reachable by alternating paths that originate in the same vertex form an interval in W. So, we search for such candidate vertices, which are called *stoppers* in the algorithm. A vertex w attributed to P_{k+1} is a stopper if each vertex in V attributed to P_{k+1} with label larger or equal to *stopper* has all its neighbors in W larger than or equal to *stopper*. In other words, $V[i].begin \geq w$, for all i such that $V[i].label \geq stopper$. If alternating paths that originate in P_{k+2} only reach vertices attributed to P_{k+1} larger or equal to the *stopper*, then no vertex attributed to P_k is reached by such paths. For this reason, the values of these stoppers, that can be determined in parallel, are also communicated to all processors. Each processor constructs an array $Stopper[k+1..p]$ containing the number of these vertices. If there is more than one candidate to be a stopper in a processor, it is enough to communicate the one which is the smallest.

Summarizing, each processor P_k, after receiving the array $V[a..b]$ sorted by labels, determines locally and communicates to all processors:

1. the value of *min_rel*, the number of the smallest vertex of W reachable by alternating paths that originate in vertices attributed to P_k;
2. the value of *min_abs*, the number of the smallest vertex of W with neighbors in vertices attributed to P_k;
3. the value of *stopper*, the number of the smallest vertex of W attributed to P_k that is candidate to be the endpoint of an interval of reachable vertices.

These values are computed locally by **Procedure Preprocess** given below.

Procedure Preprocess

Input: An array $V[a..b]$ sorted by labels, representing a convex bipartite graph $G = (V, W, E)$ without isolated vertices.

Output: The values of *stopper*, *min_rel*, and *min_abs*.

```
{Computation of $stopper$ and $min\_abs$}
$stopper := V[b].label + 1$
$ind := b + 1$
$i := b$
while $i \geq a$ do
    $candidate := V[i].begin$
    while $i \geq a$ and $V[i].label \geq candidate$ do
        $candidate := \min\{candidate, V[i].begin\}$
        $i := i - 1$
    if $candidate = V[i+1].begin$ and $candidate \geq V[a].label$ and $V[i+1].M \neq 0$ then
        $stopper := candidate$
        $ind := i + 1$
$min\_abs := \min_{i \in [a,b]}\{V[i].begin\}$
{Computation of $min\_rel$}
$i := ind - 1$
$min\_rel := +\infty$
while $i \geq a$ do
    if $V[i].M \neq 0$ then {a free vertex is found}
        $candidate := V[i].begin$
        repeat
            $candidate := \min\{candidate, V[i].begin\}$
            $i := i - 1$
        until $i < a$ or $V[i].label < candidate$
        if $candidate < V[a].label$ or ($V[a].M = 0$ and $candidate = V[a].label$) then
            $min\_rel := candidate$
    else
        $i := i - 1$
Return $stopper$, $min\_rel$ and $min\_abs$
```

In our comments above, we considered alternating paths beginning in vertices attributed to P_{k+1} and P_{k+2}. However, observe that using the information collected in the **Procedure Preprocess**, each processor P_k is able to detected the existence of alternating paths reaching vertices attributed to P_k that originate in vertices attributed to some other processor P_q, with $q > k$. This is done by the **BSP/CGM Algorithm**. Processor P_k initially constructs the arrays $Min\_Rel[k+1..p]$, $Min\_Abs[k+1..p]$, and $Stopper[k+1..p]$. The arrays $Min\_Rel[k+1..p]$ and $Stopper[k+1..p]$ are inspected backwards to identify the existence of a free vertex in V attributed to some processor P_q, with $q > k$, which originates alternating paths. Whenever alternating paths are detected, P_k searches $Min\_Abs[k+1..q]$ and $Stopper[k+1..q]$ backwards to find out where those paths end, updating the value of *min_reach* accordingly. If no alternating path reaches vertices attributed to P_k, *min_reach* will end up with some value larger than $V[b].label$. Otherwise, a dummy vertex ($V[b+1]$) is added to simulate alternating paths that originate in other processors.

To finish the **BSP/CGM Algorithm**, the sequential procedures **Procedure MIS** and **Procedure Union** are called to determine the arrays I_V^k and I_W^k representing intervals of independent vertices. At the end of the algorithm, as usual for BSP/CGM algorithms, the output is distributed among the processors. The maximum independent set is given by $I_V \cup I_W$, where $I_V = \bigcup_k (I_V^k)$ and $I_W = \bigcup_k (I_W^k)$.

BSP/CGM Algorithm

Input: An array $V[1..n]$ representing a convex bipartite graph $G = (V, W, E)$ without isolated vertices. The processor P_k, $1 \leq k \leq p$, receives the array $V[a..b]$ and an integer m, where $a = (k-1)\lceil n/p \rceil + 1$, $b = \min\{n, k\lceil n/p \rceil\}$, and $m = |W|$.

Output: Each processor P_k determines arrays I_V^k and I_W^k representing intervals of vertices of a maximum independent set.

```
for all processor $P_k$ do
  for $i := a$ to $b$ do
    if $V[i].M \neq 0$ then $V[i].label := V[i].M$
    else $V[i].label := V[i].end$
In parallel, sort array $V$ according to field $V[i].label$. Ties are broken in such a way that
matched vertices come first.
for all processor $P_k$ do
  Call Procedure Preprocess to determine $stopper$, $min\_rel$, and $min\_abs$
  Communicate $stopper$, $min\_rel$ and $min\_abs$ to processor $P_{k+1}, \ldots, P_p$.
  Receive the messages and construct arrays $Min\_rel[k..p]$, $Min\_abs[k..p]$ and
  $Stopper[k..p]$.
  $i := p$
  $min\_reach := +\infty$
  while $i > k$
    $min\_reach := \min\{Min\_rel[i], min\_reach\}$
    while $i > k$ and $min\_reach < Stopper[i]$ do
      $min\_reach := \min\{min\_reach, Min\_Abs[i]\}$
      $i := i - 1$
    $i := i - 1$
  for $i := a$ to $b$ do
    $V[i].begin = \max\{V[a].label, V[i].begin\}$
  if $min\_reach \leq V[b].label$ then
    Create a new vertex $V[b+1]$ with
    $V[b+1].begin := \max\{min\_reach, V[a].label\}$,
    $V[b+1].end := V[b].end$,
    $V[b+1].M = 0$, and
    $V[b+1].label := V[b].label$
    Call Procedure MIS with input $V[a..b+1]$ to obtain arrays $V_R^k$, $W_R^k$, and integer $j$
    Remove vertex $V[b+1]$ from $V_R^k$
  else
    Call Procedure MIS with input $V[a..b]$ to obtain arrays $V_R^k$, $W_R^k$, and integer $j$
  if $k > 1$ then
    $c := V[a].label$
    Communicate $V[a].label - 1$ to processor $P_{k-1}$
  else
    $c = 1$
  if $k < p$ then Receive integer $d$ from processor $P_{k+1}$
  else $d := m$
  Call Procedure Union with input $V_R^k$, $W_R^k$, $j$, $c$, and $d$ to obtain independent sets $I_V^k$
  and $I_W^k$.
Return $I_V^k$ and $I_W^k$.
```

We now analyze the time complexity of the **BSP/CGM Algorithm**. Recall that to obtain this result, we assume that $n \geq p^2$, what is true in practical applications. First we comment on the complexity of sorting the vertices of V, which is done in Line 5 of the algorithm. Let $T_p(n, m, p)$ be the time of local computation to sort n integers in the range $[1, m]$ using p processors. Chan and Dehne [4] describe a BSP/CGM algorithm for the case that $m = O(n^c)$ for some positive constant c. The algorithm runs in time $T_p(n, m, p) = O(n/p)$ of local computation. In the case that there is no bound on m, the algorithm of Goodrich [10] runs in time $T_p(n, m, p) = O(n \log n/p)$ of local computation. In both algorithms the number of communication rounds is a constant and the total size of the messages sent and received is $O(n/p)$.

Let us now analyze the rest of the algorithm. The labeling of the vertices can be done in time $O(n/p)$. The **Procedure Preprocess** runs in time $O(n/p)$. In a round of communication, in Line 8, messages of total size $O(p)$ are distributed. The loop of Line 12 can be done in time $O(p)$. The running time of **Procedure Preprocess** called either in Line 26 or in Line 29 is $O(n/p)$. The same holds for **Procedure Union** called in Line 37.

Therefore, Algorithm BSP/CGM uses a constant number of communication rounds and runs in $O(n/p) + T_p(n, m, p)$ time of local computation. In each round, each processor sends and receives messages of total size $O(n/p)$. This implies a BSP algorithm that uses a constant number of supersteps with $O(gn/p)$ communication cost and $O(n/p) + T_p(n, m, p)$ local computation.

5 Concluding Remarks

In this work we have presented a sequential and a BSP/CGM algorithm for finding a maximum independent set in a convex bipartite graph. The input to our algorithms is a convex bipartite graph G and a greedy maximum matching of G.

Using p processors, the coarse grained algorithm requires a constant number of communication rounds in which each processor sends and receives messages of total size $O(n/p)$, and, when $m = O(n^c)$ for some constant c, $O(n/p)$ local computation time, assuming that $n \geq p^2$. This implies a BSP algorithm with constant number of supersteps, $O(gn/p)$ communication cost and $O(n/p)$ local computation.

Acknowledgements The authors would like to thank an anonymous referee for helpful comments and important suggestions that were very helpful to improve the readability of this paper.

References

1. Booth, K., Lueker, G.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Comput. Syst. Sci. **13**, 335–379 (1976)
2. Bose, P., Chan, A., Dehne, F., Latzel, M.: Coarse grained parallel maximum matching in convex bipartite graphs. In: 13th International Parallel Processing Symposium (IPPS'99), pp. 125–129 (1999)
3. Caceres, E., Chan, A., Dehne, F., Prencipe, G.: Coarse grained parallel algorithms for detecting convex bipartite graphs. In: Proc. 26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000), Konstanz, Germany (2000)
4. Chan, A., Dehne, F.: A note on coarse grained parallel integer sorting. Parallel Process. Lett. **9**, 533–538 (1999)

5. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. McGraw–Hill, New York (1990)
6. Czumaj, A., Diks, K., Przytycka, T.: Parallel maximum independent set in convex bipartite graphs. Inf. Process. Lett. **59**, 289–294 (1996)
7. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel geometric algorithms for coarse grained multicomputers. In: 9th Annual ACM Symposium on Computational Geometry, pp. 289–307 (1993)
8. Dekel, E., Sahni, S.: A parallel matching for convex bipartite graphs and applications to scheduling. J. Parallel Distrib. Comput. **1**, 185–205 (1984)
9. Glover, F.: Maximum matching in a convex bipartite graph. Nav. Res. Logist. Q. **14**, 313–316 (1967)
10. Goodrich, M.: Communication efficient parallel sorting. In: 28th Annual ACM Symposium on Theory of Computing (STOC'96), pp. 247–256 (1996)
11. Kuhn, H.: The Hungarian method for the assignment problem. Nav. Res. Logist. Q. **2**, 83–97 (1955)
12. Lipski, W., Preparata, F.: Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. Acta Inform. **15**, 329–346 (1981)
13. Steiner, G., Yeoman, J.: A linear time algorithm for maximum matchings in convex, bipartite graphs. Comput. Math. Appl. **31**(12), 91–96 (1996)
14. Valiant, L.: A bridging model for parallel computation. Commun. ACM **33**, 103–111 (1990)