Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 34 (2008) 47–62

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Coarse grained parallel algorithms for graph matching ☆

Albert Chan [a], Frank Dehne [b,*], Prosenjit Bose [b], Markus Latzel [c]

[a] Department of Mathematics and Computer Science, Fayetteville State University, Fayetteville, NC, USA
[b] School of Computer Science, Carleton University, Ottawa, Ontario, Canada
[c] Palomino Systems Inc., Toronto, Canada

Received 2 August 2006; received in revised form 23 August 2007; accepted 20 November 2007
Available online 28 November 2007

Abstract

Parallel graph algorithm design is a very well studied topic. Many results have been presented for the PRAM model. However, these algorithms are inherently fine grained and experiments show that PRAM algorithms do often not achieve the expected speedup on real machines because of large message overheads. In this paper, we present coarse grained parallel graph algorithms with small message overheads that solve the following standard graph problems related to graph matching: finding maximum matchings in convex bipartite graphs, and finding maximum weight matchings in trees. To our knowledge, these are the first efficient parallel algorithms for these problems that are designed for standard commercial parallel machines such as off-the-shelf processor clusters.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Graph matching; Parallel algorithm; Coarse grained parallel computing; Convex bipartite graph; Unrooted tree

1. Introduction

Parallel graph algorithm design is a very well studied topic. The goal is to use parallel hardware for the solution of very large scale graph problems. An abundance of literature exists on parallel graph algorithms for the PRAM (Parallel Random Access Machine) model [1]. See e.g. [42] for a survey. However, speedup results for theoretical PRAM algorithms do not necessarily match the speedups observed on real machines [2,43]. Given sufficient slackness in the number of processors, Valiant's BSP (Bulk Synchronous Parallel) approach [44] simulates PRAM algorithms optimally on distributed memory parallel systems. Valiant points out, however, that one may want to design algorithms that utilize local computations and minimize global operations [45,44]. The BSP approach requires that g (the "gap", defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor) is low, or fixed,

☆ Research partially supported by the Natural Sciences and Engineering Research Council of Canada.
* Corresponding author.
E-mail addresses: achan@uncfsu.edu (A. Chan), frank@dehne.net (F. Dehne), jit@scs.carleton.ca (P. Bose), markus@palominosys.com (M. Latzel).
URLs: http://faculty.uncfsu.edu/achan (A. Chan), http://www.dehne.net (F. Dehne).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.11.004

even for increasing number of processors. Gerbessiotis and Valiant [24] themselves describe the circumstances where PRAM simulations cannot be performed efficiently: if the factor g is high. Unfortunately, this is true for most currently available multiprocessors. That is, PRAM simulations are, in general, not efficient for currently available multiprocessors, in particular off-the-shelf processor clusters. PRAM simulations on clusters are severely handicapped by increasing message overheads due to the above mentioned “gap”. The parallel algorithms presented in this paper solve this problem for two graph problems related to graph matching.

We use a more practical version of the BSP model, referred to as the *coarse grained multicomputer* (CGM) model [21,22,18,20,17]. It is comprised of a set of p processors $P_1, \ldots, P_p$ with $O\left(\frac{n}{p}\right)$ local memory per processor and an arbitrary communication network (n refers to the total input data size). All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single h-relation with $h = O\left(\frac{n}{p}\right)$, i.e. each processor sends $O\left(\frac{n}{p}\right)$ data and receives $O\left(\frac{n}{p}\right)$ data. We require that all information sent from a given processor to another processor in one communication round is packed into one long message, thereby minimizing the message overhead. A CGM computation/communication round corresponds to a BSP superstep with communication cost $g\frac{n}{p}$ (plus the above “packing requirement”). Finding an optimal algorithm in the coarse grained multicomputer model is equivalent to minimizing the number of communication rounds as well as the total local computation time and total message size. In particular, it is important to reduce the number of communication rounds to a constant or to a slowly growing function of p that is independent of n. For example, $\log p$ and $\log^2 p$ are such slowly growing functions (and essentially constant in practice). As shown in [6,9], a number of communication rounds that is independent of n leads to parallel algorithms with good speedup in theory *and* practice because it leads to a good amortization of message overhead. When n grows, the number of messages remains unchanged and only the message size increases. Thus, the total message overhead remains unchanged but the message overhead per data item decreases. PRAM simulations do, in general, not have this property. The number of communication rounds is a function of n (e.g. $\log n$ or $\log^2 n$) and the number of messages and message overhead grow with increasing n, leading to inefficiency.

In this paper, we continue our earlier work in [19] and present coarse grained parallel graph algorithms with $O(\log p)$ and $O(\log^2 p)$ communication rounds, respectively, for solving the following standard graph problems related to graph matching:

- finding maximum matchings in convex bipartite graphs, and
- finding maximum weight matchings in trees.

In Section 4 we present a CGM algorithm for maximum matching in convex bipartite graphs with computation time $O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p}\log p)$ and $O(\log p)$ communication rounds. Refer to Table 1 for a definition of our notation. In Section 5 we present a CGM algorithm for maximum weight matching in trees. The computation time of this algorithm is $O(\frac{n}{p}\log p)$ and the number of communication rounds is $O(\log^2 p)$. Preliminary versions of the above were first presented in [4,8]. Table 1 summarizes our results.

To our knowledge, these are the first efficient parallel algorithms for these problems that are designed for standard commercial parallel machines such as off-the-shelf processor clusters. Simulating the respective PRAM algorithms on processor clusters leads to methods where the number of communication rounds is a function of n ($\log n$ and $\log^2 n$, respectively) and the number of messages and message overhead grow with

Table 1
Summary of results

	Local computation time	Number of communication rounds	Obtained speedup
Maximum matching in convex bipartite graphs	$O\left(T_s\left(\frac{n_A}{p}, \frac{n_B}{p}\right) + \frac{n_A}{p}\log p\right)$	$O(\log p)$	$O\left(\frac{p}{\log p}\right)$
Maximum matching in trees	$O\left(\frac{n}{p}\log p\right)$	$O(\log^2 p)$	$O\left(\frac{p}{\log^2 p}\right)$
Notation	n = size of input graph		
	p = number of processors	T_s = sequential computation time for the respective problem	n_A, n_B = sizes of the two vertex sets of a bipartite graph

increasing n, leading to inefficiency. Our contribution lies in reducing the number of communication rounds to $O(\log p)$ and $O(\log^2 p)$, respectively. This is a non-trivial problem, and we solve it via a careful analysis of the PRAM algorithms and new, more efficient, coarse grained parallel solutions for critical sections that led to the previous inefficiencies. As shown in [6,9], CGM algorithms for which the number of communication rounds is independent of n show good speedup in theory *and* practice. As discussed above, the important effect obtained by our methods (but not by simulated PRAM methods) is that the number of messages and message overhead stay fixed as n grows, leading to a declining message overhead per data item.

In the following Section 2 we recall some background about the BSP and CGM models as well as some basic graph notation. The subsequent sections present our main results.

2. Preliminaries

2.1. BSP model

The Bulk Synchronous Parallel (BSP) model was proposed by Valiant [45]. A BSP machine consists of p processors. Each processor has an unspecified amount of local memory. There is no global memory. The processors are connected through a network and they communicate with each other by sending and receiving messages. BSP algorithms consist of a sequence of supersteps. In each superstep, the processors operate independently performing local computation and global communication by sending and receiving messages, but the messages sent in one superstep can only be received in the next superstep. At the end of each superstep, a barrier synchronization is performed which keeps all processors synchronized.

The analysis of BSP algorithms is based on two parameters: g and L. L is an upper bound on the latency, or delay, incurred in communicating a message from its source processor/memory module to its target processor/memory module. g is the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth. Since all messages in a superstep can be grouped into a large message, if a superstep consists of w local computation and h messages being sent and received, then the time required for one superstep is $T_{\text{superstep}} = w + gh + L$. For practical purposes, we assume that the running time is dominated by g, and therefore L can be ignored.

2.2. CGM model

The coarse grained multicomputer (CGM) model, proposed by Dehneet al. [18], is a variant of Valiant's BSP model [45]. A coarse grained multicomputer $\text{CGM}(n,p)$ consists of p processors $P_1, \ldots, P_p$, where each processor has $O\left(\frac{n}{p}\right)$ local memory. Here N refers to the total memory capacity of the CGM. The processors can communicate with each other either by sending and receiving messages over the network, or by being connected to shared memory. If the processors are connected through shared memory, the size of the shared memory should be large enough for an h-Relation (see below). That is, the size of global memory should be $\Omega(n)$. CGM algorithms usually make the assumption that the problem size is *much* larger than the number of processors (i.e., $n \gg p$). A common criterion in many algorithms is that $\frac{n}{p} \geqslant \Omega(p)$. In some algorithms, this assumption is generalized to $\frac{n}{p} \geqslant \Omega(p^{\frac{1}{\epsilon}})$, where ϵ is a parameter describing the scalability of an algorithm. The scalability measures how well the algorithm behaves when the CGM gradually degenerates into a fine-grained machine.

A CGM algorithm consists of alternating local computation and global communication rounds. In the local computation rounds, only local computation on local data is allowed. In each communication round, a single h-Relation (with $h = O\left(\frac{n}{p}\right)$) is routed. The h-Relation operation is the basic communication operation used in CGM algorithms. It performs an "all-to-all" operation for each processor. During an h-Relation operation, every processor can send out messages of size $h = O\left(\frac{n}{p}\right)$, and receive messages of size $h = O\left(\frac{n}{p}\right)$. The individual sizes of messages being sent to and received from each processor need not be the same, but the total size of all messages being sent to or received from any processor cannot exceed the prescribed bounds.

A CGM algorithm is essentially a BSP algorithm with additional "packing" and "coarse grained" requirements. It is very easy to convert CGM algorithms into BSP algorithms. Every CGM h-Relation and CGM round correspond to a BSP superstep with communication cost $g\frac{n}{p}$.

Lemma 1. *A CGM algorithm with c communication rounds corresponds to a BSP algorithm with communication cost $c \times g\frac{n}{p}$, where n is the input size and p is the number of processors.*

The goal of CGM algorithm design is to minimize both the local computation and the number of communication rounds. It will be ideal if we can achieve O(1) communication rounds and $T(n,p) = \frac{T_s(n)}{p}$. From a practical point of view, i.e. for implementing this model on commercial multiprocessors, it is very important that the problem size is not a parameter for the number of communication rounds. This is usually achievable, but for many problems, it is very difficult or even impossible to reduce the number of rounds down to O(1), and we have to settle for $O(\log p)$ or $O(\log^2 p)$ rounds. Since p is usually fixed or grows very slowly in practice, and $\log p$ is a slowly growing function, the number of communication rounds is then essentially a fixed constant for most practical arrangements. This is important because empirical studies show that the number of communication rounds is the most important parameter influencing the observed running time.

2.3. Graph terminology

A graph $G = (V,E)$ is an ordered pair of sets V and E, where $E \subseteq V \times V$. It is weighted if every edge in E is associated with a numerical weight.

A bipartite graph $G = (A,B,E)$ is defined as an undirected graph $G = (V,E)$ such that $V = A \cup B$ and $A \cap B = \emptyset$. Moreover, $\forall e = (a,b) \in E$, either $a \in A$ and $b \in B$ or vice versa.

A convex bipartite graph is a bipartite graph $G = (A,B,E)$ with an ordering $B = (b_1, b_2, \ldots, b_{n_B})$ such that $\forall a \in A$, if $(a,b_i) \in E$ and $(a,b_j) \in E$ $(i \leqslant j)$ then $(a,b_k) \in E$ for all $i \leqslant k \leqslant j$.

A path in a graph g is an ordered list of edges $P = (e_1, e_2, \ldots, e_n)$ such that for each $e_k = (u_k, v_k)$, we have $v_k = u_{k+1}$. If $u_1 = v_n$, the path is called a cycle. A simple path is a path with no repeated internal vertex. Similarly, a simple cycle is a cycle with no repeated vertex. Note that a simple cycle is also a simple path.

A tree is a connected (sub)graph with no cycle. We sometimes call this a free tree to distinguish it from a rooted tree. A (free) forest is a graph which consists only of (free) trees.

A matching M in a graph $G = (V,E)$ is a subset of E such that no two edges in M are incident to the same vertex. A matching M' in a graph $G = (V,E)$ is maximum if for any matching M in g, $|M'| \geqslant |M|$.

3. Maximum matching on graphs: problem overview

Matching is an important graph problem. It has many applications and has been extensively studied. In this paper, we study the problem of computing maximum matchings in parallel (on the CGM model) for the following graphs:

- Convex bipartite graphs;
- Unrooted (free) trees.

The problem of finding a maximum matching in a bipartite graph or a convex bipartite graph is a classic and well-studied problem [23,10,28,38]. The case of convex bipartite graphs has several interesting applications as outlined in [23,38]. A particularly interesting industrial application for matching parts with products was described in [38]. Recall that the vertices of a bipartite graph or a convex bipartite graph can be divided into two disjoint groups. Here we assume that the size of these two groups are n_A and n_B, respectively. Sequential solutions for maximum matching in bipartite graphs and convex bipartite graphs with time complexities $O(n_A^{\frac{5}{2}})$ and $O(n_A + \alpha(n_B))$ were described in [28] and [38], respectively, where $\alpha(\ldots)$ is a very slowly growing function related to the inverse Ackermann function.

3.1. Previous parallel algorithms

The vast majority of previous results on parallel matching are for the shared memory PRAM model of computation. A PRAM algorithm for maximum matching in convex bipartite graphs requiring $O(\log^2 n_A)$ time and $\frac{n_A}{2}$ processors was presented in [23]. Pawagi [41] has presented a PRAM algorithm for finding a maximum weight matching in a tree of size n. His algorithm requires $O(\log^2 n)$ computation time and $O(n)$ processors. Andrews et al. presented an algorithm for maximum matching in interval graphs [3]. Dahlhaus et al. presented algorithms on matchings for various graphs [14,12,13,15,16,11]. Goldberg et al. have a sublinear-time parallel algorithm for matching [25]. Grover published an article on bipartite graph matching [27]. Kao et al. presented a nearly optimal parallel algorithm for maximum matching in planar graphs [29]. Karpinski et al. also have published on parallel algorithms for graph matching [30]. Kelsen presented an algorithm for finding matching in expander graphs [31] and an optimal parallel algorithm for maximum matching [32]. Klein presented matching algorithms for chordal graphs [34–37]. Moitra et al. presented an algorithm for maximum matching on interval graphs [39]. Nakanishi et al. gave a parallel algorithm for matching in general graphs [40].

For mesh connected computers, Kim et al. provided an optimal parallel matching algorithm for convex bipartite graphs [33]. Wu et al. studied the message complexity of finding maximum matchings in general graphs [46] for distributed systems.

3.2. Our contributions in this paper

In this paper, we study parallel solutions for matching problems on coarse grained parallel processor clusters and present coarse grained parallel CGM algorithms for maximum matching in convex bipartite graphs and unrooted trees. An outline of our CGM model is given in Section 2.2.

In our previous work we have shown that in order to obtain efficient coarse grained parallel algorithms in practice, it is important to minimize the number of communication rounds (see e.g. [9,21,6,22]). All of the above prior methods for parallel graph matching are fine grained (i.e. they assume $\Omega(n)$ processors) and require a large number of communication rounds when executed on a processor cluster.

The CGM introduces the additional difficulty that we require a small number of communication rounds in order to provide good performance in practice. A straight forward simulation of the above PRAM algorithms would result in $\Omega(\log n)$ communication rounds which is not effective in practice. Our contribution is to reduce this to $O(\log p)$ communication rounds which is a small constant for typical cluster installations.

CGM algorithms with a small fixed number of communication rounds on a fixed size cluster have the important property that the total number of messages sent/received is fixed, regardless of the problem size. All prior algorithms require a number of messages that increases with the problem size, leading to increased message overhead. For our CGM algorithms, if the problem size is increased, the number of messages stays constant but the message sizes increase. Message overhead stays constant and the overhead per data item is actually decreased.

4. Maximum matching in convex bipartite graphs

Our solution for solving maximum matching problem for convex bipartite graphs on a CGM follows the general template of the PRAM algorithm in [23] which reduces the problem to maximum interval assignment. The CGM introduces the additional difficulty that we require a small number of communication rounds in order to provide good performance in practice. A straight forward simulation of the PRAM algorithm in [23] would result in $O(\log n)$ communication rounds which is not effective in practice. Our contribution is to reduce this to $O(\log p)$ communication rounds which is a small constant for typical cluster installations and has been shown in our previous work to be efficient in practice (e.g. [9,21,6,22]).

We now discuss the reduction to maximum interval assignment.

Definition 1. Given a set of intervals $I = \{I_1, \ldots, I_{n_A}\}$, each of which represents an integer range, i.e. $I_i = (l_i, r_i)$, a maximum interval assignment consists of assigning, for a maximum number of intervals, one integer label to each interval such that the integer for each interval is within the interval's range and no two intervals are assigned the same integer.

For a given convex bipartite graph $G = (A, B, E)$, let $I(G)$ be the set of $|A|$ intervals containing for each $a \in A$ an interval $I_i = (l_i, r_i)$ where l_i and r_i are the smallest and largest ranks, in B, of all element $b \in B$ with $(a, b) \in E$.

Observation 1

The maximum matching problem for a convex bipartite graph G can be reduced to finding a maximum interval assignment in *$I(G)$*.

Proof. In the matching M, every vertex $u_i \in A$ can be incident to at most one edge which leads to a vertex in B. Finding such an edge is equivalent to finding an integer within the range of the interval associated to I_i. □

In Section 4.1, we first present an algorithm that solves a special case: all intervals start at the same left endpoint. This algorithm will be used in our solution for the general case which is presented in Section 4.2.

4.1. Special case: all intervals start at the same point

The following Algorithm 1 solves the special case where all intervals start at the same left endpoint. We first sort all intervals by right endpoint and assign each processor a "controlled range" where it can locally assign labels to its locally stored intervals. Clearly, this will not necessarily result in a maximum interval assignment. We then introduce a global data exchange and adjustment phase (Algorithm 2) which creates a globally optimal assignment.

Algorithm 1. All intervals have the same left endpoint $l_i = l,\ i = 1, \ldots, n_A$.

Input: A set of n_A intervals $I = \{I_1, \ldots, I_{n_A}\}$ with their associated left and right endpoints $I_i = (l_i, r_i),\ i = 1 \ldots n_A$, distributed over a p processor CGM with $\frac{n_A}{p}$ intervals per processor. Note that we assume $\frac{n_A}{p} \geqslant \Omega(p)$.
Output: A maximum interval assignment of I.

(1) All intervals are sorted by their right endpoints using CGM parallel integer sort [7,26].
(2) Each processor $P_i,\ i = 1, \ldots, p-1$, determines the largest right endpoint, e_i, received in Step 1 and sends it to the next processor P_{i+1}.
(3) Each processor $P_i,\ i = 2, \ldots, p$, sets $s_i = e_{i-1} + 1$. The first processor sets $s_1 = l$. We call (s_i, e_i) the controlled range of P_i. This is the range of integers that the processor can use to label intervals.
(4) Each processor $P_i,\ i = 1, \ldots, p$, temporarily changes the left endpoints of its intervals to s_i and then solves the modified problem locally using a sequential algorithm (essentially sequential integer sort).
(5) Each processor $P_i,\ i = 1, \ldots, p$, calculates how many labels in the controlled range are left unused (a_i), and how many intervals have not yet received an integer label (b_i). These two numbers (per processor) are broadcast to all processors (in one h-Relation). This is possible since $\frac{n}{p} \geqslant \Omega(p)$.
(6) Based on the data received in the previous step, each processor can now calculate where to request labels from and where to send unused labels to. See Algorithm 2 for details.
(7) Using the results of Algorithm 2, each processor selects unused labels and sends them to the processors that need them.
(8) Upon receiving the needed labels, each processor can now assign them to the respective intervals.

We now explain in more detail Steps 6 and 7, and how they relate to Algorithm 2. In Step 6 of Algorithm 1, the sequential Algorithm 2 presented below is executed by every processor. Every processor uses as input the array $a = (a_1, \ldots, a_p)$ representing the numbers of free (unused) labels in each of the p processors, and the array $b = (b_1, \ldots, b_p)$ representing for each processor the number of intervals with no labels yet. Both arrays were received in Step 5. Algorithm 2 calculates for each processor two arrays $get = (get_1, \ldots, get_p)$ and $give = (give_i, \ldots, give_p)$ representing the numbers of labels to be received and provided from the respective processors. In Step 7 of Algorithm 1, the free labels are then exchanged accordingly.

The sequential Algorithm 1 uses two pointers i and j, pointing to arrays a and b, respectively, and scanning them from left to right. Processor i can provide labels to processor j if $i < j$ and $a_i, b_j > 0$. Furthermore, in order to maximize the label assignment, processor i will first provide as many as possible (and required) labels to processor $i+1$, then $i+2$, and so on. This implies a double pointer scan, left to right.

Algorithm 2. Sequential computation for Step 6 in Algorithm 1, executed by each processor P_k, $1 \leqslant k \leqslant p$.

```
Input: An array $a = (a_1, \ldots, a_p)$ representing the numbers of free (unused) labels in the $p$ processors. An
  array $b = (b_1, \ldots, b_p)$ representing the number of intervals with no labels yet in each of the $p$ processors.
Output: An array $get = (get_1, \ldots, get_p)$ representing numbers of labels that are to be sent to processor $P_k$ by
  the other processors. An array $give = (give_i, \ldots, give_p)$ representing numbers of labels that are to be
  received from processor $P_k$ by the other processors.
Initialize the arrays get and give to 0.
$i \leftarrow 1$; {pointer to array $a$}
$j \leftarrow 2$; {pointer to array $b$}
while $((i \leqslant p)\&(j \leqslant p))$ do
  if $(i = j)$ then
    $j++$;
  end if
  while $((a_i \neq 0)\&(j \leqslant p))$ do
    if $(a_i \geqslant b_j)$ then
      if $(P_k = i)$ then
        $give_j \leftarrow b_j$; do
      end if
      if $(P_k = j)$ then
        $get_i \leftarrow b_j$;
      end if
      $a_i \leftarrow a_i - b_j$;
      $b_j \leftarrow 0$;
      $j++$;
    else
      if $(P_k = i)$ then
        $give_j \leftarrow a_i$;
      end if
      if $(P_k = j)$ then
        $get_i \leftarrow a_i$;
      end if
      $b_j \leftarrow b_j - a_i$;
      $a_i \leftarrow 0$;
    end if
  end while
  $i++$;
end while
```

Lemma 2. *The sequential time complexity of Algorithm 2 is $O(p)$.*

Proof. The variable i counts from 1 to $p-1$, and j counts from 2 to p. Since both variables are incremented independently, the total time complexity of the algorithm is $O(p)$. □

Lemma 3. *The integer label assignment produced by Algorithm 1 is maximum.*

Proof. It is easy to see that the label assignment based on the order of the right endpoints is maximum. Let A_0 be such an assignment. Let A_1 be the label assignment produced by Algorithm 1. Since A_0 is maximum, $|A_0| \geqslant |A_1|$. We compare assignments A_1 and A_0. We first remove the intervals that are not assigned labels in both assignments. If the remaining intervals all have labels assigned in both assignments, then we have $|A_0| = |A_1|$ and the lemma follows.

If the kth interval (l'_k, r'_k) (with respect to the right endpoints) is assigned a label in A_0 but not in A_1, then the first $k-1$ intervals must be assigned labels $l, l+1, \ldots, l+k-2$ in A_0 and the kth interval must be assigned label $l+k-1$. This also means $r'_k \geqslant l+k-1$.

Now consider A_1. If the first $k-1$ intervals are also assigned labels $l, l+1, \ldots, l+k-2$, then label $l+k-1$ will be available for the kth interval. If some of the first $k-1$ intervals are not assigned labels in the range $l, l+1, \ldots, l+k-2$, then there will be a "hole" in this range that can be used for the kth interval.

Hence, for every interval with a label assigned in A_0, there is a label assigned to that interval in A_1. Therefore, $|A_1| \geqslant |A_0|$. Thus, the label assignment obtained from Algorithm 1 is maximum. □

Theorem 1. *Algorithm 1 finds a maximum label assignment for n_A intervals with the same left endpoints stored on a p processor CGM with $\frac{n_A}{p}$ local memory per processor, $\frac{n_A}{p} \geqslant \Omega(p)$, in $O(1)$ communication rounds with $O\left(\frac{n_A}{p}\right)$ local computation.*

Proof. The correctness of the algorithm is shown in Lemma 3. The communication rounds required in each step of Algorithm 1 is $O(1)$. Hence the total number of communication rounds needed is $O(1)$. In each round, the total message size per processor is $O\left(\frac{n_A}{p}\right)$. Step 1 uses CGM integer sort [7,26] which requires $O(1)$ communication rounds and $O(\frac{n_A}{p})$ local computation. The local computation of Step 6 is $O(p)$ (from Lemma 2). For all other steps, it is either $O(1)$ or $O\left(\frac{n_A}{p}\right)$. Therefore, the total local computation per processor is $O\left(\frac{n_A}{p}\right) + O(p) = O\left(\frac{n_A}{p}\right)$ since $\frac{n_A}{p} \geqslant \Omega(p)$. □

4.2. The general case

In order to solve the maximum interval assignment problem for arbitrary intervals, we now combine Algorithm 1 with the binary tree method presented in [23].

Algorithm 3. General case where the intervals can have different left endpoints.

Input: A set of n intervals $I = \{I_1, \ldots, I_{n_A}\}$ with their associated left and right endpoints $I_i = (l_i, r_i)$, $i = 1 \ldots n_A$, distributed over a p processor CGM with $\frac{n_A}{p}$ intervals per processor. Note that we assume that $\frac{n_A}{p} \geqslant \Omega(p)$.

Output: A maximum interval assignment of I.

(1) All intervals are sorted by their left endpoints using CGM parallel integer sort [7,26]. (In case of a tie, intervals are compared by their right endpoints.)
(2) Intervals with the same left endpoints are combined into groups. All groups that are stored completely within a processor are merged into a single group. Let γ be the number of groups. Note that γ is at most $2p+1$.
(3) Each group is assigned a *controlled range* (l_i, r_i), $i = 1 \ldots \gamma$, where l_i is equal to the smallest left endpoint of that group and $r_i = l_{i+1} - 1$, $i = 1 \ldots \gamma - 1$. Let r_γ be the largest right endpoint of the intervals.
(4) Using a sequential algorithm, each processor solves the problem for interval groups that are completely within the processor. Remove the label of all those intervals that received a label outside their group's controlled range. Classify the intervals using one of the following three types: matchable (M) for all labelled intervals; to-be-determined (T) for all non-labelled intervals that extend beyond the rightmost label given by that processor; and unmatchable (U) for all remaining intervals. Remove all unmatchable intervals.

(5) Using Algorithm 1, solve the problem for interval groups that cross a processor boundary (for each such group in isolation). Remove the label of all intervals that received a label outside their group's controlled range. Classify the intervals using one of the following three types: matchable (M) for all labelled intervals; to-be-determined (T) for all non-labelled intervals that extend beyond the rightmost label given by that processor; and unmatchable (U) for all remaining intervals. Remove all unmatchable intervals.

(6) Merge the intervals in adjacent groups. Let M_{L}, T_{L} be the intervals from the left group, let M_{R}, T_{R} be the intervals from the right group, and let $(l_{\mathrm{L}}, r_{\mathrm{L}})$ and $(l_{\mathrm{R}}, r_{\mathrm{R}})$ be the controlled ranges of the left and right groups, respectively. Using Algorithm 1, solve the problem for $T_{\mathrm{L}} \cup M_{\mathrm{R}}$ over the range $(l_{\mathrm{R}}, r_{\mathrm{R}})$. Let the resulting sets of matchable and to-be-determined intervals be M_n and T_n. For the combined group, set $M = M_{\mathrm{L}} \cup M_n$ and $T = T_n \cup T_{\mathrm{R}}$. The controlled range of the combined group is $(l_{\mathrm{L}}, r_{\mathrm{R}})$.

(7) Repeat Step 6 $\mathrm{O}(\log\gamma)$ times until a single group is obtained.

(8) Using a reverse process of the previous steps, using $\mathrm{O}(\log\gamma)$ iterations, redistribute the intervals back into the γ groups obtained at the end of Step 3. In each splitting phase, let M be the group we are splitting and let M_{L} and M_{R} be the two corresponding components. Let V be the set of intervals that have a left endpoint smaller than the starting value of the controlled range of M. Note that V can be empty. Let W be the union of V and M_{L}. W should be distributed to M_{L} and the rest to M_{R}. However, the controlled range of M_{L} may not allow the entire W to be assigned to M_{L}. Hence, we assign only the first $\min(|W|, l\mathrm{R}\text{–}\mathrm{L}|)$ intervals to M_{L} and the remainder to M_{R}.

(9) Step 8 is repeated until γ groups are obtained.

(10) Using a sequential algorithm, each processor solves the problem for its interval groups over each group's controlled range. For groups that span over more than one processor, adjust the left end points to the starting values of the controlled range and then apply Algorithm 1.

Theorem 2. *Algorithm 3 solves the label assignment problem in* $\mathrm{O}(\log p)$ *communication rounds and* $\mathrm{O}\left(T_{\mathrm{s}}\left(\frac{n_A}{p}, \frac{n_B}{p}\right) + \frac{n_A}{p}\log p\right)$ *local computation where* $T_{\mathrm{s}}(n_A, n_B)$ *is the running time of the optimal sequential algorithm for the problem.*

Proof. The correctness of the split/merge scheme follows from [23]. Step 6 is executed $\mathrm{O}(\log\gamma)$ times. In Step 6, we invoke Algorithm 1 which requires $\mathrm{O}(1)$ communication rounds. Since $\gamma \leqslant 2p+1$, the total number of communication rounds is $\mathrm{O}(\log p)$. The local computation time is dominated by the $\mathrm{O}(1)$ executions of the sequential algorithm on each processor and the linear local time in each of the $\mathrm{O}(\log\gamma)$ iterations. □

Theorem 2 implies a BSP algorithm with local computation $\mathrm{O}\left(T_{\mathrm{s}}\left(\frac{n_A}{p}, \frac{n_B}{p}\right) + \frac{n_A}{p}\log p\right)$ and communication cost $\mathrm{O}\left(g\frac{n_A}{p}\log p\right)$. We can calculate the speedup for this algorithm as follow. Let $T_{\mathrm{s}}(n_A, n_B) = \mathrm{O}(n_A + \alpha(n_B))$ be the running time of the best sequential algorithm[38], and let $T_p(n_A, n_B, p)$ be the running time of our algorithm. Since α is a very slowly growing function, for practical purpose, we can reduce T_{s} to $T_{\mathrm{s}}(n_A, n_B) = \mathrm{O}(n_A)$ and T_p to

$$T_p(n_A, n_B, p) = \mathrm{O}\left(\frac{n_A}{p} + \frac{n_A}{p}\log p\right) + \mathrm{O}\left(g\frac{n_A}{p}\log p\right)$$

Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore

$$s = \frac{T_{\mathrm{s}}(n_A, n_B)}{T_p(n_A, n_B, p)} = \frac{cn_A}{d\left(\frac{n_A}{p} + \frac{n_A}{p}\log p\right) + kg\frac{n_A}{p}\log p} = \frac{p}{g\log p}\left(\frac{c}{\frac{d}{g\log p} + \frac{d}{g} + k}\right) = \Omega\left(\frac{p}{g\log p}\right)$$

This theorem implies an algorithm for computing the maximum matching of a convex bipartite graph with the same number of communication rounds and local computation.

5. Maximum weighted tree matching

In this section we consider the problem of finding a maximum weight matching in a rooted or unrooted tree. That is, given a tree $T = (V, E)$ and a weight $w_i \geqslant 0$ for each edge $e_i \in E$, we want to find a matching $M \subset E$ of T such that the total weight $w = \sum_{e_i \in M} w_i$ is maximized.

As mentioned before, Pawagi [41] presented a PRAM algorithm for maximum weight tree matching. For a tree of size n, his algorithm requires $O(\log^2 n)$ computation time and $O(n)$ processors.

We will present a parallel CGM algorithm for maximum weight tree matching that requires $O(\log^2 p)$ communication rounds with $O\left(\frac{n}{p}\log p\right)$ local computation and $O\left(\frac{n}{p}\right)$ memory per processor. Our algorithm assumes that the local memory per processor, $\frac{n}{p}$, is larger than $\Omega(p^\epsilon)$ for some fixed $\epsilon > 0$. This assumption is true for all commercially available multiprocessors. Our results imply a BSP algorithm with $O(\log^2 p)$ supersteps, $O(g\log(p)\frac{n}{p})$ communication time, and $O(\log(p)\frac{n}{p})$ local computation time.

Pawagi's PRAM algorithm [41] would require $O(\log^2 n)$ communication rounds when implemented on a cluster. The main contribution of this section is to present a CGM algorithm that reduces the number of communication rounds to $O(\log^2 p)$. As discussed in Section 3.2, this is a significant efficiency improvement for solving this problem on a cluster.

The following is an outline of our CGM algorithm for maximum weight tree matching. For the algorithm description we assume w.l.o.g. a tree T that is rooted at an arbitrary node. An unrooted tree can be converted into a rooted tree in $O(\log n)$ communication rounds by using the connected component algorithm in [19].

Algorithm 4. Maximum weight tree matching.

Input: A tree T with n vertices and $n-1$ edges evenly distributed (in arbitrary order) over a p-processor coarse grained multicomputer (CGM).
Output: The same tree T with all matched edges marked. All edges along the maximum gain path are marked. The maximum gain value and a boolean representing whether the root is free or not.

```
(1) Mark all vertices as free. If $p = 1$, solve the problem sequentially and return.
(2) Compute the centroid $c$ of $T$ using Algorithm 5. Re-root $T$ at $c$ using Algorithm 6. Partition $T$ into subtrees $T_1, T_2, \ldots, T_k$, where $k$ is the total in-degree of $c$. Each subtree $T_i$ is assigned a unique key $k_i$. Let $v_i$ be the root of $T_i$.
(3) For each subtree $T_i$, let $n_i$ be the size of the subtree. Assign $\lceil \frac{n_i}{n} p \rceil$ processors to recursively solve the problem on the subtree. For this, we need to redistribute the tree and move the nodes to their respective processors via a parallel CGM integer sort by key $k_i$ [7].
(4) For each subtree $T_i$, let $e_i$ be the edge connecting $c$ and $T_i$, and let its weight be $w_i$. Compute the maximum gain in $T_i$ using Algorithm 7. Compute the value
    $q_i \leftarrow \{ \begin{array}{ll} w_i & \text{if root of } T_i \text{ is free} \\ w_i + \text{maximum gain in } T_i & \text{otherwise} \end{array} .$
(5) Select a $j$ such that $q_j$ is the maximum.
(6) If $q_j \leqslant 0$ then mark $c$ as free and go to Step 9.
(7) If the root $v_j$ of $T_j$ is free then mark $(c, v_j)$ as matched, mark $c$ as not-free and go to Step 9.
(8) Mark the edge $(c, v_j)$ as matched. Invert the matching for all edges with key $k_j$. That is, for each edge $e$ with key $k_j$, mark the edge matched if it was not matched originally and mark it unmatched if it was matched originally. Mark $c$ as not-free.
(9) Re-root $T$ to $r$ using Algorithm 6.
```

While our algorithm follows the general structure of Pawagi's PRAM algorithm [41], the implementation details are very different. For example, Pawagi's algorithm includes steps such as finding the centroid of a tree, re-rooting a tree to a given node, and finding a maximum gain alternating path of a tree. These steps are straight forward for the PRAM but non-trivial for the CGM.

The main contribution of our work lies in the study of these individual steps and how they can be implemented on a CGM with a small number of communication rounds. Our main result is a parallel maximum weight tree matching algorithm which is suitable to be executed on PC clusters. In the following sections, we present our CGM algorithms for finding the centroid of a tree, re-rooting a tree and finding a maximum gain alternating path of a tree. These methods are also interesting in their own right, and probably useful for other coarse grained parallel graph algorithms. We conclude with an analysis of Algorithm 4 in Section 5.4.

5.1. Algorithm 5: partitioning a tree via its centroid

We are using divide-and-conquer to solve the maximum weight matching problem. In order for divide-and-conquer to work efficiently, we partition the tree such that the subproblems are guaranteed to shrink by at least a constant factor.

In this section, we present an algorithm to do such partitioning efficiently for rooted trees. The result can be easily generalized to unrooted trees.

Definition 2. The centroid of a tree T is a vertex c that minimizes the size of the largest subtree in the forest generated by deletion of c from T.

Note that for an n-vertex tree, the size of the largest subtree in the forest generated by the deletion of the centroid is less than or equal to $\frac{n}{2}$ [41].

Algorithm 5 (*Centroid of a tree*).

Input: A tree T with n vertices and $n-1$ edges, rooted at r. All nodes and edges are evenly distributed (in arbitrary order) over a p-processor coarse grained multi-computer.
Output: A vertex c which is the centroid of T.

(1) Compute an Euler tour of T [5]. See Fig. 1 for an illustration of Euler Tour traversal.
(2) Apply list ranking [5]. That is, calculate for each vertex its distance from the root r along the Euler tour. From these distances, we can easily calculate the size of the subtrees for each vertex. Among these subtrees, the subtree with maximum size for the vertex is selected.
(3) Each processor determines which vertex has the smallest maximum subtree. The selected vertex is sent to Processor P_1.
(4) Processor P_1 selects and outputs the vertex that has the smallest maximum subtree.

Theorem 3. *Algorithm 5 calculates the centroid of a rooted tree using* $O(\log p)$ *communication rounds,* $O\left(\frac{n}{p}\log p\right)$ *local computation, and* $O\left(\frac{n}{p}\right)$ *storage per processor.*

Proof. The correctness of the algorithm follows immediately from the definition of the centroid. All steps are within the stated bounds. □

5.2. Algorithm 6: re-rooting a tree

In this section, we describe a CGM algorithm for re-rooting a tree, i.e. changing the root of a tree and updating all edge directions accordingly.

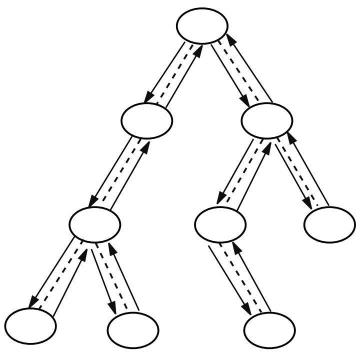

Fig. 1. Illustration of the Euler tour traversal of a tree.

Algorithm 6 (*Re-rooting a tree*).

Input: A tree T with n vertices and $n-1$ edges, rooted at r. Each vertex v in T has a pointer $parent(v)$ to its parent. $parent(r)=\emptyset$. A vertex $u \neq r$ in T (the new root). All nodes and edges are evenly distributed (in arbitrary order) over a p-processor coarse grained multi-computer.
Output: A tree T' that is topologically equivalent to T but rooted at u.

(1) Find an Euler tour E of T. [5]
(2) Perform list ranking along E [5]. Each edge e in T will receive two ranks, r_{e_1} and r_{e_2}. Assume $r_{e_1} < r_{e_2}$.
(3) Let the ranks for the edge $(u, parent(u))$ be r_1 and r_2. Broadcast these two ranks to all processors.
(4) Each processor for each edge e, compares r_{e_1}, r_{e_2} with r_1 and r_2. Mark the edge e if $r_{e_1} \leqslant r_1$ and $r_{e_2} \geqslant r_2$. Note that all marked edges form a path from u to r.
(5) Each processor for each vertex v checks if the edge $(v, parent(v))$ is marked. If so, send v to the processor storing $parent(v)$.
(6) Every vertex v' that received a new vertex v'' sets its parent $parent(v) = v''$.
(7) The vertex u sets its parent to $parent(u) = \emptyset$.

Theorem 4. *Using a p processor CGM, Algorithm* 6 *re-roots an n-node tree in* $O(\log p)$ *communication rounds with* $O\left(\frac{n}{p}\log p\right)$ *local computation and* $O\left(\frac{n}{p}\right)$ *memory per processor.*

Proof. The correctness of the algorithm lies on the properties of the Euler tour. As shown in Fig. 2, the edges can be divided into three zones. The edges on the "left" of the path $(u,\ldots,r)$ (zone 1 in Fig. 2) will have $r_{e_1} \leq r_{e_2} \leq r_1$. The edges on the "right" of the path $(u,\ldots,r)$ (zone 2 in Fig. 2) will have $r_2 \leq r_{e_1} \leq r_{e_2}$. The edges "below" the path $(u,\ldots,r)$ (zone 3 in Fig. 2) will have $r_1 \leq r_{e_1} \leq r_{e_2} \leq r_2$. The edge on the path $(u,\ldots,r)$ (and only the edges on the path) will have $r_{e_1} \leqslant r_1$ and $r_{e_2} \geqslant r_2$. Thus, Step 4 will mark all the edges on the path $(u,\ldots,r)$, and Step 6 will reverse all the parent pointers along the path. Therefore, Algorithm 6 can re-root a tree.

Step 1 requires $O(1)$ communication rounds, $O\left(\frac{n}{p}\right)$ local computation and local storage. Step 2 requires $O(\log p)$ communication rounds, $O\left(\frac{n}{p}\log p\right)$ local computation and $O\left(\frac{n}{p}\right)$ local storage. Step 3 requires $O(1)$ communication rounds. Step 4 and 6 require no communication, $O\left(\frac{n}{p}\right)$ local computation and local storage. Step 5 requires $O(1)$ communication rounds. Step 7 requires no communication and $O(1)$ local computation. □

5.3. *Algorithm 7: maximum gain alternating path*

We start this section with several definitions.

Definition 3. Given a graph $G=(V,E)$ and a matching $M \subseteq E$, a node $v \in V$ is free if and only if $\forall u \in V$, $(u,v) \notin E$ or $(u,v) \notin M$.

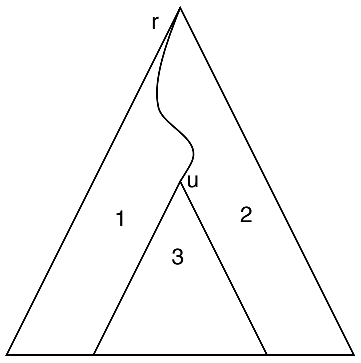

Fig. 2. Re-rooting a tree.

Definition 4. A path $P = (v_{i_1}, v_{i_2}, \dots, v_{i_n} = v)$ in T is said to be an alternating path with respect to a matching M if every odd numbered edge on this path is in M and every even numbered edge is not in M or vice versa.

Definition 5. The gain of an alternating path P, denoted by $G(P)$, is the difference between the sum of the weights of the edges of P that are not in M and the sum of the weights of the edges of P that are in M.

An alternating path with positive gain can be used to obtain a new matching by removing from M all edges that are in P and adding the unmatched edges in P to M. Observe that the resulting matching is at least as large as the original one.

Algorithm 7 (*Maximum gain proper alternating path*).

Input: A weighted tree T rooted at r, a key k, and a maximum matching M in T. All nodes and edges are evenly distributed (in arbitrary order) over a p-processor coarse grained multi-computer.
Output: A vertex u in T such that the path from r to u forms a proper alternating path with maximum gain in T. All edges along the path $(r, \dots, u)$ are marked with the key k.

```
(1) Let $parent(v)$ be the parent of $v$, and $w_v$ be the weight of the edge $(v, parent(v))$. For each node $v$ in the tree calculate $g(v)$ defined as follows (local computation only):
    – if $v$ is the root $r$, then $g(r) = 0$.
    – if $(v, parent(v)) \in M$, then $g(v) = -w_v$.
    – if $(v, parent(v)) \notin M$, then $g(v) = w_v$.
(2) Except for the root $r$, each node $v$ checks the values of $g(v)$ and $g(parent(v))$. If both values are positive, delete the edge $(v, parent(v))$ (local computation only).
(3) Using the CGM list ranking algorithm in [5], determine for each node $v$ its distance to the root $r$ as well as the sum of the $g(v)$ along the path $P = (r, \dots, v)$. (See proof of Theorem 5 for detail.) Due to the definition of the $g(v)$, the sum of the $g(v)$ along the path $P$ is equal to the gain $G(P)$. Remove all nodes that are not able to reach the root $r$.
(4) Build the modified Euler tour $E_m$ of $T$ where all nodes $v$ that are not leaves in $T$ and $(v, parent(v)) \notin M$ are removed. (use the CGM pointer jumping algorithm in [5]).
(5) Using a CGM partial sum algorithm [9,6] on $E_m$ with maximum operator with respect to the $G((r, \dots, v))$ values of the nodes in $E_m$, determine the node $u$ that has the maximum gain.
(6) Similar to Step (1) to (4) of Algorithm 6, mark all edges along the path $(r, \dots, u)$ with the supplied key $k$.
```

In Algorithm 7, if r is not free, then the edge $(r, u) \in M$ must be in the maximum alternating path. This is because M is maximum. An alternating path with positive gain can be used to obtain a new matching by removing from M all edges that are in P and adding the unmatched edges in P to M. Observe that the resulting matching is larger than the original one.

Theorem 5. *Algorithm 7 computes the maximum gain proper alternating path in* $O(\log p)$ *communication rounds with* $O\left(\frac{n}{p}\log p\right)$ *local computation and* $O\left(\frac{n}{p}\right)$ *memory per processor.*

Proof. The correctness of the algorithm relies on whether we can find the correct gain $G((r, \dots, v))$ for each node v in Step 3. This can be achieved by building an Euler tour of the tree and performing list ranking. More precisely, we build an Euler tour by defining for each tree edge e_i between a node v_i and it's parent $parent(v_i)$ two directed edges $e_{i_1} = (parent(v_i), v_i)$ and $e_{i_2} = (v_i, parent(v_i))$. We then assign the value $g(v_i)$ to e_{i_1} and the value $-g(v_i)$ to e_{i_2}. See Fig. 3 for an illustration. Observe that in Fig. 3, for each node v the values of the edges in the subtree rooted at v cancel out to zero. Therefore, a partial sum along the Euler tour shown in Fig. 3 will compute the gain $G((r, \dots, v))$ for each node v. Thus, in Step 5, the resulting path will be the maximum gain alternating path. We use the CGM list ranking algorithm in [5] to compute the partial sum along the Euler tour shown in Fig. 3 using $O(\log p)$ communication rounds.

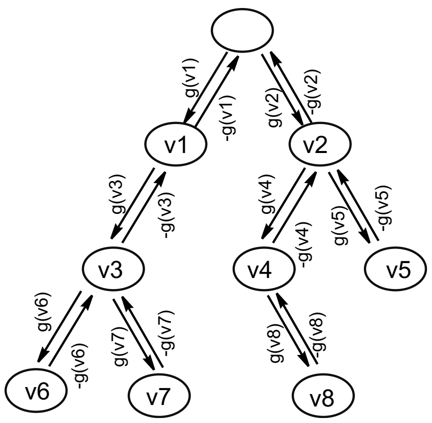

Fig. 3. Gain computation through Euler tour list ranking.

Steps 1, 2 and 4 can be performed using $O(1)$ communication rounds, $O\left(\frac{n}{p}\right)$ computation and storage. Step 3 can be performed using $O(\log p)$ communication rounds, $O\left(\frac{n}{p}\log p\right)$ computation and $O\left(\frac{n}{p}\right)$ storage. Steps 5 and 6 can be executed using $O(\log p)$ communication rounds, $O\left(\frac{n}{p}\right)$ computation and storage. □

5.4. Analysis of Algorithm 4

Theorem 6. *Algorithm 4 computes the maximum weight matching for the tree T using* $O(\log^2 p)$ *communication rounds, with* $O\left(\frac{n}{p}\log p\right)$ *local computation and* $O\left(\frac{n}{p}\right)$ *memory per processor.*

Proof. The main step is Step 4 where we select an edge e_i such that adding e_i to the matching will result in a maximum increase in the total weight in the matching. Note that in Step 3, the matching in each subtree T_i has been maximized. Therefore, if the root of T_i is free, then by adding e_i to the matching, the total increase will be w_i. If the root of T_i is not free then, by adding e_i to the matching and flipping the edges along the maximum gain alternating path, the total increase of the weight will be w_i plus the maximum gain in T_i. At the end of the algorithm, the resulting matching will always be maximized. Therefore, Algorithm 4 is correct.

Step 2 will guarantee that the size of the maximum subtree will be at most one half of the original tree. Thus, after at most $O(\log p)$ recursions, each subtree will fit into one processor and we can apply the sequential algorithm to solve the subproblem. Step 1 will be executed once for each subtree, which will take no communication and use $O\left(\frac{n}{p}\right)$ local computation and storage. Steps 2, 4 and 9 can be done using $O(\log p)$ communication rounds, $O\left(\frac{n}{p}\log p\right)$ computation and $O\left(\frac{n}{p}\right)$ local storage. Steps 5 and 8 can be done using $O(1)$ communication rounds, $O\left(\frac{n}{p}\right)$ computation and local storage. Steps 6 and 7 can be done without communication and with $O\left(\frac{n}{p}\right)$ computation and local storage. Therefore, the total bounds for each recursion will be $O(\log p)$ communication rounds, $O\left(\frac{n}{p}\log p\right)$ computation and $O\left(\frac{n}{p}\right)$ storage. Since the size of each subproblem is guaranteed to be at most half the size of the original problem, the theorem follows. □

Theorem 6 implies a BSP algorithm with local computation $O\left(\frac{n}{p}\log p\right)$ and communication cost $O(g\frac{n}{p}\log^2 p)$. We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm

$$T_p(n,p) = \mathrm{O}\left(\frac{n}{p}\log p\right) + \mathrm{O}\left(g\frac{n}{p}\log^2 p\right)$$

is the running time of our algorithm. Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore

$$s = \frac{T_\mathrm{s}(n)}{T_p(n,p)} = \frac{cn}{\frac{dn}{p}\log p + \frac{kgn}{p}\log^2 p} = \frac{p}{g\log^2 p}\left(\frac{c}{\frac{d}{g\log p} + k}\right) = \Omega\left(\frac{p}{g\log^2 p}\right)$$

6. Conclusion

The following table summarizes the results obtained in this paper. Note that n_A and n_B are the sizes of the two vertex sets in the convex bipartite graph matching problem, n is the input size in the tree matching problem, and p is the number of processors available. Finally $T_\mathrm{s}(n_A, n_B)$ is the performance of the best sequential convex bipartite graph matching algorithm available.

Algorithm	Local computation	Communication rounds	Speedup
Maximum matching in convex bipartite graphs	$\mathrm{O}\left(T_\mathrm{s}\left(\frac{n_A}{p}, \frac{n_B}{p}\right) + \frac{n_A}{p}\log p\right)$	$\mathrm{O}(\log p)$	$\Omega\left(\frac{p}{g\log p}\right)$
Maximum matching in trees	$\mathrm{O}\left(\frac{n}{p}\log p\right)$	$\mathrm{O}(\log^2 p)$	$\Omega\left(\frac{p}{g\log^2 p}\right)$

References

[1] M. Allen, B. Wilkinson, Parallel Programming – Techniques and Applications using Networked Workstations and Parallel Computers, Prentice-Hall, Englewood Cliffs, New Jersy, 1999.

[2] R. Anderson, L. Snyder, A comparison of shared and nonshared memory models of computation, IEEE 79 (4) (1993) 480–487.

[3] M.C. Andrews, M.J. Atallah, D.Z. Chen, D.T. Lee, Parallel algorithms for maximum matching in interval graphs, in: Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95), IEEE Computer Society Press, Santa Barbara, CA, USA, 1995, pp. 84–92.

[4] P. Bose, A. Chan, F. Dehne, M. Latzel, Coarse grained parallel maximum matching in convex bipartite graphs, in: Proceedings of the 13th International Parallel Processing Symposium (IPPS'99), 1999, pp. 125–129.

[5] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, S.W. Song, Efficient parallel graph algorithms for coarse grained multicomputers and bsp, in: P. Degano, R. Gorrieri, A. MarchettiSpaccamela (Eds.), Proceedings of the ICALP'97 – 24th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 1256, Springer-Verlag, 1997, pp. 390–400.

[6] A. Chan, F. Dehne, R. Taylor, Implementing and testing cgm graph algorithms on pc clusters and shared memory machines, International Journal of High Performance Computing Applications 19 (1) (2005) 81–97.

[7] A. Chan, F. Dehne, A note on coarse grained parallel integer sorting, Parallel Processing Letters 9 (4) (1999) 533–538.

[8] A. Chan, F. Dehne, A coarse grained parallel algorithm for maximum weight matching in trees, in: Proceedings of the 12th IASTED International Conference Parallel and Distributed Computing and Systems (PCDS 2000), 2000, pp. 134–138.

[9] A. Chan, F. Dehne, CGMgraph/CGMlib: implementing and testing CGM graph algorithms on PC clusters, in: Proceedings of the 10th EuroPVM/MPI, LNCS, vol. 2840, 2003, pp. 117–125.

[10] F. Clover, Maximum matching in convex bipartite graphs, Naval Research Logistics Quarterly 14 (1967) 313–316.

[11] Elias Dahlhaus, Marek Karpinski, Andrzej Lingas, A Parallel Algorithm for Maximum Matching in Planar Graphs, Technical Report TR-89-018, ICSI, 1989.

[12] Elias Dahlhaus, Marek Karpinski, On the Parallel Complexity of Matching for Chordal and Path Graphs, Technical Report 8512 – CS, Institut fúr Informatik der Universitát Bonn, 1987.

[13] Elias Dahlhaus, Marek Karpinski, Parallel Complexity for Matching Restricted to Degree Defined Graph Classes, Technical Report 859 – CS, Institut fúr Informatik der Universitát Bonn, 1987.

[14] Elias Dahlhaus, Marek Karpinski, Parallel Construction of Perfect Matchings and Hamiltonian Cycles on Dense Graphs, Technical Report 8518 – CS, Institut fúr Informatik der Universitát Bonn, 1987.

[15] Elias Dahlhaus, Marek Karpinski, On the Computational Complexity of Matching on Chordal and Strongly Chordal Graphs, Technical Report TR-94-043, ICSI, 1994.

[16] Elias Dahlhaus, Marek Karpinski, The Parallel and Sequential Complexity of Matching on Chordal and Strongly Chordal Graphs, Technical Report 85129 – CS, Institut fúr Informatik der Universitát Bonn, 1995.

[17] F. Dehne, X. Deng, P. Dymond, A. Fabri, A.A. Kokhar, A randomized parallel 3d convex hull algorithm for coarse grained multicomputers, in: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Dig. Library, 1995, pp. 27–33.

[18] F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable parallel computational geometry for coarse grained multicomputers, in: Proceedings of the ACM Symposium on Computational Geometry (SoCG), ACM Dig. Library, 1993, pp. 298–307.

[19] F. Dehne, A. Ferreira, E. Caceres, S.W. Song, A. Roncato, Efficient parallel graph algorithms for coarse grained multicomputers and BSP, Algorithmica 33 (2) (2002) 183–200.

[20] F. Dehne, C. Kenyon, A. Fabri, Scalable and architecture independent parallel geometric algorithms with high probability optimal time, in: Proceedings of the IEEE Symposium on Parallel and Distributed Processing (SPDP), IEEE Computer Society Press, 1994, pp. 586–593.

[21] F. Dehne, Guest editor's introduction: special issue on coarse grained parallel algorithms, Algorithmica 24 (3/4) (1999) 173–176.

[22] F. Dehne, Guest editor's introduction: special issue on coarse grained parallel algorithms for scientific applications, Algorithmica 45 (3) (2006) 263–267.

[23] E. Dekel, S. Sahni, A parallel matching algorithm for convex bipartite graphs and applications to scheduling, Journal of Parallel and Distributed Computing 1 (1984) 185–205.

[24] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, in: Proceedings of the Scandinavian workshop on algorithm theory, Lecture Notes in Computer Science, vol. 621, Springer Verlag, 1992, pp. 1–18.

[25] Andrew V. Goldberg, Serge A. Plotkin, Pravin M. Vaidya, Sublinear-time parallel algorithms for matching and related problems, Journal of Algorithms 14 (2) (1993) 180–213.

[26] M. Goodrich, Communication efficient parallel sorting, in: ACM Symposium on Theory of Computing (STOC), 1996.

[27] L.K. Grover, Fast parallel algorithms for bipartite matching, in: E. Balas, G. Cornuéjols, R. Kannan (Eds.), Proceedings of the 2nd Integer Programming and Combinatorial Optimization Conference, Carnegie Mellon University, Pittsburgh, PA, 1992, pp. 367–384.

[28] J. Hopkroft, R. Karp, An $n^{\frac{5}{2}}$ algorithm for maximum matching in bipartite graphs, SIAM Journal in Computing 2 (1973) 225–231.

[29] Ming Kao, Marek Karpinski, Andrzej Lingas, Nearly Optimal Parallel Algorithm for Maximum Matching in Planar Graphs, Technical Report 8549 – CS, Institut fúr Informatik der Universitát Bonn, 1990.

[30] Marek Karpinski, Wojciech Rytter, Fast parallel algorithms for graph matching problems, Oxford Lecture Series in Mathematics and its Applications, vol. 9, Clarendon Press, 1998.

[31] Pierre Kelsen, Fast parallel matching in expander graphs, in: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, Velen, FRG, 1993, pp. 293–299.

[32] Pierre Kelsen, An optimal parallel algorithm for maximal matching, Information Processing Letters, vol. 52, Elsevier, Amsterdam, 1994, pp. 223–228.

[33] Myung-Ho Kim, Chang-Sung Jeong, Myung-Soo Kim, An optimal parallel matching algorithm for a convex bipartite graph on a mesh-connected computer, in: Howard Jay Siegel (Ed.), Proceedings of the 8th International Symposium on Parallel Processing, IEEE Computer Society Press, Cancún, Mexico, 1994, pp. 229–233.

[34] P. Klein, Efficient Parallel Algorithms for Chordal Graphs, in: 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 150–161.

[35] P. Klein, Efficient Parallel Algorithms for Planar, Chordal, and Interval Graphs, PhD Thesis, MIT, 1988.

[36] P. Klein, Efficient parallel algorithms for chordal graphs, in: Proceedings of the 29th Symposium on the Foundations of Computer Science (FOCS), 1989, pp. 150–161.

[37] P. Klein, Parallel algorithms for chordal graphs, in: Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers, 1993, pp. 341–407.

[38] W. Lipski, F. Preparata, Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems, Acta Informatica 15 (1981) 329–346.

[39] A. Moitra, R.C. Jo hnson, A parallel algorithm for maximum matching on interval graphs, in: International Conference on Parallel Processing, Vol. 3: Algorithms and Applications, The Pennsylvania State University, 1989, pp. 114–120.

[40] Yoshinori Nakanishi, Kuniaki Uehara, Parallel organization algorithm for graph matching and subgraph isomorphism detection, in: Setsuo Arikawa, Hiroshi Motoda (Eds.), Proceedings of the 1st International Conference on Discovery Science (DS-98), LNAI, vol. 1532, Springer, Fukuoka, Japan, 1998, pp. 407–408.

[41] S. Pawagi, Parallel algorithm for maximum weight matching in trees, in: International Conference on Parallel Processing, Pennsylvania State University Press, University Park, Penn State University, PA, USA, 1987, pp. 204–206.

[42] J. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan and Kaufmatin Publishers, 1993.

[43] L. Snyder, Type architectures, shared memory and the corollary of modest potential, Annual Review of Computational Science 1 (1986) 289–317.

[44] L. Valiant et al., General purpose parallel architectures, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, MIT Press, Elsevier, 1990, pp. 943–972.

[45] L. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990).

[46] M.W. Wu, M.C. Loui, An efficient distributed algorithm for maximum matching in general graphs, Algorithmica 5 (1990).