Coarse Grained Parallel Algorithms for Detecting *Convex* Bipartite Graphs *

Extended Abstract

E. Caceres † **A. Chan** ‡ **F. Dehne** § **G. Prencipe** ¶

Abstract

In this paper, we present parallel algorithms for the *coarse grained multicomputer* (CGM) and *bulk synchronous parallel computer* (BSP) for solving two well known graph problems: (1) determining whether a graph G is *bipartite*, and (2) determining whether a bipartite graph G is *convex*.

Our algorithms require $O(\log p)$ and $O(\log^2 p)$ communication rounds, respectively, and linear sequential work per round on a CGM with p processors and N/p local memory per processor, $N{=}|G|$. The algorithms assume that $\frac{N}{p} \geq p^\epsilon$ for some fixed $\epsilon > 0$, which is true for all commercially available multiprocessors. Our results imply BSP algorithms with $O(\log p)$ and $O(\log^2 p)$ supersteps, respectively, $O(g \log(p) \frac{N}{p})$ communication time, and $O(\log(p) \frac{N}{p})$ local computation time.

Our algorithm for determining whether a bipartite graph is convex includes a novel, coarse grained parallel, version of the *PQ tree* data structure introduced by Booth and Lueker. Hence, our algorithm also solves, with the same time complexity as indicated above, the problem of testing the consecutive-ones property for $(0,1)$ matrices as well as the chordal graph recognition problem. These, in turn, have numerous applications in graph theory, DNA sequence assembly, database theory, and other areas.

1 Introduction

In this paper, we study the problem of detecting bipartite graphs and convex bipartite graphs. That is, given an arbitrary graph G, determine whether G is a *bipartite* graph and, given a bipartite graph G, determine whether G is a *convex* bipartite graph. Bipartite and convex bipartite graphs are formally defined as follows.

Definition 1 *A graph $G = (V, E)$ is a bipartite graph if V can be partitioned into two sets A and B such that $A \cap B = \emptyset$, $A \cup B = V$ and $E \subseteq ((A \times B) \cup (B \times A))$. A bipartite graph G is also denoted as $G = (A, B, E)$.*

Definition 2 *A bipartite graph $G = (A, B, E)$ is a convex bipartite graph if there exists an ordering $(b_1, b_2, \cdots, b_{|B|})$ of B such that, for all $a \in A$ and $1 \leq i < j \leq |B|$, if $(a, b_i) \in E$ and $(a, b_j) \in E$ then $(a, b_k) \in E$ for all $i \leq k \leq j$.*

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada and by PRONEX-FINEP, Brazil.

†Dept. de Computacao e Estatistica, UFMS, Campo Grande, Brasil, edson@dct.ufms.br

‡School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, achan@scs.carleton.ca

§School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, dehne@scs.carleton.ca

¶Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy, prencipe@di.unipi.it

These, and closely related, problems has been extensively studied for the sequential [1, 16] and the shared memory (PRAM) parallel [5, 6, 12, 13, 14, 15] domain. Unfortunately, theoretical results from PRAM algorithms do not necessarily match the speedups observed on *real* parallel machines. In this paper, we present parallel algorithms that are more practical in that the assumptions and cost model used reflects better the reality of commercially available multiprocessors. More precisely, we will use a version of the BSP model, referred to as the *coarse grained multicomputer* (CGM) model. In comparison to the BSP model, the CGM [7, 8, 9, 10] allows only bulk messages in order to minimize message overhead costs. A CGM is comprised of a set of p processors $P_1, \ldots, P_p$ with $O(N/p)$ local memory per processor and an arbitrary communication network (or shared memory). All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single h-relation with $h = O(N/p)$, i.e. each processor sends $O(N/p)$ data and receives $O(N/p)$ 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. The CGM model has the advantage of producing results which correspond much better to the actual performance of implementations on commercially available parallel machines. In addition to minimizing communication and computation volume, it also minimizes important other costs like message overheads and processor synchronization.

In this paper, we present parallel CGM algorithms for detecting bipartite graphs and convex bipartite graphs. The algorithms require $O(\log p)$ and $O(\log^2 p)$ communication rounds, respectively, and linear sequential work per round. They assume that the local memory per processor, N/p, is larger than p^ϵ for some fixed $\epsilon > 0$. This assumption is true for all commercially available multiprocessors. Our results imply BSP algorithms with $O(\log p)$ supersteps, $O(g \log(p)\frac{N}{p})$ communication time, and $O(\log(p)\frac{N}{p})$ local computation time.

The algorithm for detecting bipartite graphs is fairly simple and is essentially a combination of tools developed in [3]. The larger part of this paper deals with the problem of detecting *convex* bipartite graphs. This is clearly a much harder problem. It has been extensively studied in the literature and is closely linked to the *consecutive ones* problem for $(0,1)$-matrices as well as chordal graph recognition [1, 5, 6, 12, 13, 14, 15, 16].

Our algorithm for determining whether a bipartite graph is convex includes a novel, coarse grained parallel, version of the *PQ tree* data structure introduced by Booth and Lueker [1]. Hence, our algorithm also solves, with the same time complexity as indicated above, the problem of testing the consecutive-ones property for $(0,1)$-matrices as well as the chordal graph recognition problem. These, in turn, have numerous applications in graph theory, DNA sequence assembly, database theory, and other areas. [1, 5, 6, 12, 13, 14, 15, 16]

2 Detecting *Bipartite* Graphs

In this section, we present a fairly simple CGM algorithm for detecting bipartite graphs. It is essentially a combination of tools developed in [3].

Algorithm 1 Detection of Bipartite Graphs

Input: A Graph $G = (V, E)$ with vertex set V and edge set E, $|G| = N$, stored on a CGM with p processors and $O(N/p)$ memory per processor; $N/p \geq p^{\epsilon}$ for some fixed $\epsilon > 0$. V and E are arbitrarily distributed over the memories of the CGM. **Output:** A Boolean indicating whether G is a bipartite graph and, if it is, a partition of V into two disjoint set A and B such that $E \subseteq ((A \times B) \cup (B \times A))$.

(1) Compute a spanning forest of G [3].

(2) For each tree in the forest, select one arbitrary node as the root. Apply the CGM Euler Tour algorithm in [3] to determine the distance between each node and the root of its tree. Classify the nodes into two groups: the nodes with an odd numbered distance to the root, and the nodes with an even numbered distance to the root.

(3) Each processor examines the edges stored in its local memory. If any such edge has two vertices that belong to the same group, the result for that processor is "failure"; otherwise, the result is "success".

(4) By applying CGM sort [11] to all "failure"/"success" values, it is determined whether there was any processor with a "failure" result. If there was any "failure", the graph G is *not* bipartite. Otherwise, G *is* a bipartite graph, and the two groups of vertices identified in Step 2 are the sets A and B.

Theorem 1 *Algorithm 1 detects whether $G = (V, E)$, $|G| = N$, is a bipartite graph and, if so, partitions E into sets A and B such that $E \subseteq ((A \times B) \cup (B \times A))$ in $O(\log p)$ communication rounds and $O(\frac{N}{p})$ local computation per round on a CGM with p processors and $O(\frac{N}{p})$ memory per processor, $\frac{N}{p} \geq p^{\epsilon}$ for some fixed $\epsilon > 0$.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3 Detecting *Convex* Bipartite Graphs

We now turn our attention to the problem of testing whether a given bipartite graph is a convex bipartite graph. The sequential solution, presented by Booth and Lueker [1], introduced a data structure called *PQ-tree*. Our coarse grained parallel solution will include a coarse grained parallel version of the PQ-tree. We will first review Booth and Lueker's PQ-tree definition.

3.1 PQ-Trees

A PQ-tree [1] is a tree data structure that represents a class of permissible permutations over a universal set.

Definition 3 *A tree T is a PQ-tree if every internal node of T can be classified as either a P-node or a Q-node. A P-node is an internal node that has at least 2 children, and the children can be permuted in any order. A Q-node is an internal node that has at least 3 children, and the children can only be permuted in two ways: the original order or the reverse order. The leaves of the PQ-tree are elements of a universal set $S = \{a_1, \ldots, a_n\}$, usually called the ground set.*

The order of the ground set in the PQ-tree, from left to right, is called its *frontier*. The frontier of a PQ-tree is clearly a permutation of the ground set. Given a PQ-tree T and using only permissible permutations of its internal nodes, we can generate a number of permutations of S. We will denote with $L(T)$ the set of all these permissible permutations. A PQ-tree T' is *equivalent* to T if T' can be transformed into T using only permissible permutations of the internal nodes (if $L(T')$ and $L(T)$ have the same elements).

Given a set $A \subset S$, we say that $\lambda \in L(T)$ *satisfies* A if all elements of A appear consecutively in λ. The main operation on a PQ-tree T is called *reduce*: given a *reduction* set $\mathcal{A} = \{A_1, \ldots, A_k\}$ of subsets of S and a PQ-tree T, we want obtain a PQ-tree T', if it exists, such that each permutation in $L(T')$ satisfies every A_i, $1 \leq i \leq k$.

Let $m = \Sigma_{i=1}^{k} |A_i|$ and $N = n + m$. In order to store T and $\mathcal{A}$, we require a coarse grained multicomputer with p processors and N/p local memory per processor.

Two particular PQ-trees are the *universal* and the *empty* tree: the first one has only one internal node (the root of T) and that internal node is a P-node; the second one (also called a *null* PQ-tree) is used to represent an impossible reduction, that is when it is impossible to reduce a PQ-tree with respect to a given reduction set.

3.2 Multiple *Disjoint* Reduce Operations on a PQ-Tree (MDReduce)

In this section, we will present a coarse grained parallel algorithm for the special case of performing multiple *disjoint* reductions on a PQ-tree. We will then use this solution to develop the general algorithm in the subsequent section. More precisely, given a PQ-tree T we will first study how to perform the *reduce* operation for a set $\mathcal{A} = \{A_1, \ldots, A_k\}$ of subset of the universal set S where $A_1, \ldots, A_k$ are disjoint. We shall refer to our algorithm as *Algorithm MDReduce*. For ease of discussion, each set A_i is assigned a unique color, and we color the leaves of the PQ-tree accordingly. As shown in [13], *MDReduce* can be partitioned into the following phases: (1) pre-process the PQ-Tree (Algorithm 2); (2) process all the P-nodes (Algorithm 3); (3) process all the Q-nodes (Algorithm 4); (4) if T is not a null tree, each processor examines its nodes and relabel each orientable node (defined later) to become an R-node; (5) post-process the PQ-Tree (Algorithm 5). In the following subsections of Section 3.2, we present coarse grained parallel algorithms for each phase. The PQ-tree definitions used are from [1, 13, 14, 15].

3.2.1 Pre-Processing the PQ-Tree

The pre-processing phase extends the coloring δ of the leaves to a coloring Δ of all nodes of the PQ-tree T. For an internal node v of T, we say that a color is *complete* at v if all the leaves with that color are descendants of v. We say a color is *incomplete* at v if some, but not all, of the leaves of that color are descendants of v. We say that a color *covers* v if all the leaves below v are of that color, and that v is *uncovered* if no color covers v. Let $LCA(c)$ be the lowest common ancestor of all leaves with color c. Let $COLORS(v)$ denote the set of colors assigned to leaves that are descendents of v. Let $INC(v)$ be the set of colors which are incomplete at v. Then $INC(v) = COLORS(v)$ - $\{c$: $LCA(c)$ is a descendent of $v\}$.

Algorithm 2 Pre-Processing the PQ-Tree
Input: The original PQ-tree T.
Output: The original PQ-tree T in which each node is assigned a "coloring" Δ, or, if failure occurs, a null tree.

(1) Apply the coarse grained parallel Lowest Common Ancestor (LCA) algorithm [3].
(2) Expand T into a binary tree B, as described in Klein [13].
(3) Perform tree contraction on B[3]. For each node v_b in B, let v_p be the node in T from which v_b is created. Let w_1 and w_2 be the children of v_b. The operation for the tree contraction is $INC(v_b) = INC(w_1) \cup INC(w_2)$ - $\{c$: $LCA(c)$ is a descendent of $v_p\}$. If at any point the size of INC is more than two, stop and return a null tree.

(4) Let c_v be a new color unique to node v. Each processor, for all its nodes, v, calculates $\Delta(v) = < c_1, c_2 >$ as follows: If two colors are incomplete at v, then c_1 and c_2 are these colors. If only one color c is incomplete at v but c does not cover v, then $c_1 = c$ and $c_2 = c_v$. If one color c is incomplete at v and covers v, then $c_1 = c_2 = c$. If no color is incomplete at v, then $c_1 = c_2 = c_v$.

Lemma 1 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algorithm 2 can be completed in $O(\log p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3.2.2 Processing P-Nodes

A node v in a PQ-tree is *orientable* if it is a Q-node and the two colors in its $\Delta(v) =< c_1, c_2 >$ are different, i.e. $c_1 \neq c_2$.

For a color c, define $h_v(c) = \begin{cases} c & \text{if } c \in INC(v) \\ c_v & \text{if } c \notin INC(v) \end{cases}$

For a PQ-tree T where w_1 and w_k are the leftmost and rightmost elements, respectively, of the frontier $fr_T(v)$, let $l_T = h_v(\delta(w_1))$ and $r_T = h_v(\delta(w_k))$. If $lr_T[v] =< l_T[v], r_T[v] >$ then we use the following notation: $< a, b > \sim < a', b' >$ if $\{a, b\} = \{a', b'\}$.

Algorithm 3 Processing P-Nodes
Input: The PQ-Tree output from Algorithm 2.
Output: The original PQ-tree T in which all the P-nodes have been processed, or, if failure occurs, a null tree.

(1) If the input PQ-tree T is a null tree, return T.
(2) Each processor sets variable $FAILURE$ to $FALSE$
(3) Each processor, for each P-node v, reorder the children of v such that for each color c all children covered by c are consecutive.
(4) Each processor, for each P-node v and each color c, if there are at least two children covered by c (and at least one child not covered by c) then insert a new P-node w_c between these c-covered children and v.
(5) Each processor, for each P-node v, constructs an auxiliary graph G_v whose nodes are the children of v and where for each color c there is an edge between children v_i and v_j at which c is incomplete if v_i or v_j is covered by c, or there is no child covered by c. If any node has more than 2 neighbors, set $FAILURE$ to $TRUE$ to indicate a failure condition.
(6) Perform a multi-broadcast of the variable $FAILURE$. If any of the broadcast values is $TRUE$, return a null tree.
(7) Each processor uses list-ranking to identify the connected components of each G_v and verifies that each of these connected components is a simple path. If any of these components is a cycle, set $FAILURE$ to $TRUE$ to indicate a failure condition. We call these paths *color chains*.
(8) Perform a multi-broadcast of the variable $FAILURE$. If any of the broadcast values is $TRUE$, return a null tree.
(9) Each processor, for each color chain χ containing at least 2 nodes, chooses one of the 2 orientations of χ arbitrarily. Reorder the children of v so that the nodes of χ are consecutive, and insert a new Q-node between these nodes of v.
(10) Each processor, for each P-node v, let $S = \{v_i : v_i \text{ is a child of } v, \text{ and } INC(v_i)=\emptyset\}$. If every child of v is in S, then return. Otherwise, reorder the children of v to make S consecutive, insert a new P-node v' between v and the subset S (if $|S| > 1$), and rename v to be a Q-node.

Lemma 2 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algorithm 3 can be completed in $O(\log p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3.2.3 Processing Q-Nodes

For each Q-node v, we define an orientation $\overline{LR}(v)$ which is either $\Delta(v_i)$ or $\Delta(v_i)^R$. Note that if $\Delta(v_i) =< c_1, c_2 >$ than $\Delta(v_i)^R =< c_2, c_1 >$ For $< a, b > \sim < a', b' >$ and $a \neq b$, we define $< a, b > swap < a', b' >$ equals $TRUE$ if $< a, b > =< b', a' >$, $FALSE$ if $< a, b > =< a', b' >$. For a Q-node v, $flip$ is defined as the operation which re-orders all its children in reverse order.

Algorithm 4 Processing Q-Nodes
Input: The PQ-tree output from Algorithm 3.
Output: The original PQ-tree T in which all the Q-nodes have been processed, or, if failure occurs, a null tree.

(1) If the input PQ-tree T is a null tree, return T.
(2) Each processor sets variable $FAILURE$ to $FALSE$
(3) Each processor, for each Q-node v and children be $v_1, \ldots, v_s$, assign to each $\overline{LR}[v_i]$ either $\Delta(v_i)$ or $\Delta(v_i)^R$ such that every color in the sequence $\overline{LR}[v_1], \ldots, \overline{LR}[v_s]$ occurs consecutively, and such that $h_v(< \overline{L}[v_1], \overline{R}[v_s >]) \sim \Delta(v)$. If this is impossible, set $FAILURE$ to $TRUE$ to indicate a failure condition, otherwise, set $LR[v]$ to $h_v(< \overline{L}[v_1], \overline{R}[v_s] >)$.
(4) Perform a multi-broadcast of the variable $FAILURE$. If any of the broadcast values is $TRUE$, return a null tree.
(5) Each processor for each node v: if v is orientable, then set $OPP[v]$ to $LR[v]\ swap\ \overline{LR}[v]$, otherwise, set $OPP[v]$ to $FALSE$.
(6) Each processor for each node v: set $REV[v]$ to $\bigoplus_{u \text{ is an ancestor of } v} OPP[u]$ (Note: $\bigoplus$ denotes "exclusive-or").
(7) For each orientable node v, if $REV[v]$ is $TRUE$, then flip v.

Lemma 3 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algorithm 4 can be completed in $O(\log p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3.2.4 Post-Processing the PQ-Tree

Algorithm 5 Post-Processing the PQ-Tree
Input: The PQ-tree output from Algorithm 4, with all R-nodes renamed.
Output: Result of Algoithm *MDReduce*.

(1) If T is a null tree, return.
(2) Each processor temporarily cuts the links of its Q-nodes to their parents.
(3) Each processor performs pointer jumping for all its nodes that are children of R-nodes to determine their lowest Q-node ancestor.
(4) Each processor restores the links cut in Step 2.
(5) Each processor eliminate its R-nodes by setting the parents of their children to their lowest Q-node ancestors.

Lemma 4 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algoritm 5 can be completed using in $O(\log p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3.2.5 Analysis

Theorem 2 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algorithm* MDReduce *performs a multiple disjoint reduce for a PQ-tree T in $O(\log p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Proof. Omitted due to page restrictions; consult [4] for details. □

3.3 Multiple *General* Reduce Operations on a PQ-Tree (MReduce)

Using the coarse grained parallel *MDreduce* algorithm presented in the previous section, we will now develop coarse grained parallel algorithm for the general *MReduce* operation: given a PQ-tree T over the ground set S with n elements, perform the reduce operation for an arbitrary reduction sets $\mathcal{A} = \{A_1, \ldots, A_k\}$

Our CGM algorithm for the general *MReduce* operation consists of two phases. In the first phase, we execute $3 \log p$ times an algorithm similar to the one proposed by Klein for the PRAM [13, 14, 15]. We call this operation KLEIN-LIKE-MREDUCE$(T, \{A_1, \ldots, A_k\}, 0)$. Our contribution here is the coarse grained parallel implementation of the various steps which is, in some cases, non trivial. After the first phase, we have reduced the problem to one in which we are left with a set of smaller PQ-trees over ground sets whose size is at most n/p. Hence, each tree can be stored in the local memory of one processor. However, we can not guarantee that all the *reduction sets* of these PQ-trees do also fit in the local memory of one processor. In the second part of our algorithm, we use a merging strategy to complete the algorithm. We will refer to this phase as the *Merging Phase*.

An illustration of our general algorithm strategy is given in Figure 1.

3.3.1 First Phase: KLEIN-LIKE-MREDUCE

For a node v of a PQ-tree, $leaves_T(v)$ denotes the set of pendant leaves of v, i.e. leaves of T having v as ancestor. Let $lca_T(A)$ denote the least common ancestor in T of the leaves belonging to A. Suppose that $v = lca_T(A)$ has children $v_1, \ldots v_s$ in order. We say A is *contiguous* in T if either (1) v is a Q-node, and for some consecutive subsequence $v_p, \ldots, v_q$ of the children of v, $A = \bigcup_{p \le i \le q} leaves_T(v_i)$, or (2) v is a P-node or a leaf, and $A = leaves_T(v)$.

Suppose that E is contiguous in T. $T|E$ denotes the subtree consisting of $lca_T(E)$ and those children of $lca_T(E)$ whose descendents are in E (it is still a PQ-tree whose ground set is E). For a set A, define

$$A_i|E = \begin{cases} A_i \cap E & \text{if } A_i \cap E \neq E \\ \emptyset & \text{if } A_i \cap E = E \end{cases}$$

Let $\star_E$ denote $lca_T(E)$. T/E denotes the subtree of T obtained by omitting all the proper descendents of $lca_T(E)$ that are ancestors of elements of E (it is still a PQ-tree whose ground set is $S - E \cup \{\star_E\}$). For a set A, define

$$A_i/E = \begin{cases} A_i - E \cup \{\star_E\} & \text{if } A_i \supseteq E \\ A_i - E & \text{otherwise} \end{cases}$$

Algorithm 6 KLEIN-LIKE-MREDUCE$(T, \{A_1, \ldots, A_k\}, i)$[13, 14, 15]:

(1) If $i = 3 \log p$, return.
(2) Purge the collection of input sets A_i of empty sets. If no sets remain, return.
(3) Let n be the size of the ground set of T. If $n \leq 4$, carry out the reduction one by one. If the size of the input is smaller than the size of the local memory of the processors, than solve the problem sequentially using the Booth and Lueker's algorithm.
(4) Otherwise, let $\mathcal{A}$ be the family of (nonempty) sets A_i. Let $\mathcal{S}$ consist of the sets A_i such that $|A_i| \leq n/2$. We call such sets "small". Let $\mathcal{L}$ be the remaining, "large", sets in $\mathcal{A}$. Find the connected components of the intersection graph of $\mathcal{A}$, find a spanning forest of the intersection graph of $\mathcal{S}$, and find the intersection $\cap\mathcal{L}$ of the large sets.
(5) Proceed according to one of the following cases:
   (a) The intersection graph of $\mathcal{A}$ is disconnected. In this case, let $\mathcal{C}_1, \ldots, \mathcal{C}_r$ be the connected components of $\mathcal{A}$. For $i = 1, \ldots, r$, let E_i be the union of sets in the connected component $\mathcal{C}_i$. Call MDREDUCE to reduce T with respect to the disjoint sets $E_1, \ldots, E_r$. Next, for each $i = 1, \ldots, r$ in parallel, recursively call KLEIN-LIKE-MREDUCE$(T|E_i, \mathcal{C}_i, i+1)$.
   (b) The union of sets in some connected component of $\mathcal{S}$ has cardinality at least $n/4$. In this case, from the small sets making up this large connected component, select a subset whose union has cardinality between $n/4$ and $3n/4$. Let E be this union, and call SUBREDUCE$(T, E, \{A_1, \ldots, A_k\}, i)$.
   (c) The cardinality of the intersection of the large sets is at most $3n/4$. In this case, from the large sets choose a subset whose intersection has cardinality between $n/4$ and $3n/4$. Let E be this intersection, and call SUBREDUCE$(T, E, \{A_1, \ldots, A_k\}, i)$.
   (d) The other case do not hold. In this case, let E be the intersection of the large sets, and call SUBREDUCE$(T, E, \{A_1, \ldots, A_k\}, i)$.

In the full version of this paper [4], we show how to implement the above on a coarse grained multicomputer with p processors and $O(\frac{n}{p})$ storage per processor in $O(\log p)$ communication rounds. The non trivial parts are Step 4, Step 5b, the computation of E, T/E, and $T|E$, as well as the SUBREDUCE operation. The latter involves another operation called GLUE. Due to page restrictions, we can not present this part of our result in the extended abstract. Instead, we give one example which shows the coarse grained parallel computation of the set E in Step 5(b) of Algorithm 6.

Algorithm 7 Computation of E.
Input: The set $\mathcal{S}$ and the spanning forest of its intersection graph.

(1) In order to find a connected component $\mathcal{C}$ in the spanning forest of $\mathcal{S}$, such that the union of its sets has cardinality at least $n/4$, order all the components according to the labeling given by the coarse grained parallel *spanning forest* algorithm [3].
(2) Sort each component with respect to the values of its elements and mark as "valid" only one element per distinct value.
(3) Sort again with respect to the components' labels. Compute the cardinality of the union of the elements of each component (that is the *size* of each component), with a prefix-sum computation, counting only the "valid" elements. (Hence, we do not count twice the elements with same values and compute correctly the cardinality of the union.)
(4) If a processor finds a component whose size is $\geq n/4$, then it broadcasts the label of this component. Otherwise it broadcast a "not-found" message.
(5) If everybody sent "not-found", go to step 4(c) of MREDUCE algorithm. Otherwise, among all the labels received in the previous step, choose as $\mathcal{C}$ the component with the smallest label.
(6) For each of the sets comprising $\mathcal{C}$, compute the distance in the spanning tree (from the root) using the coarse grained parallel Euler-tour technique [3].
(7) Sort the sets according to distance, and let $B_1, \ldots, B_s$ be the sorted sequence. Sort each sets with respect to the values of its elements and mark as "valid" only one element per distinct value. Sort

the sets again, according to distance, and let $\hat{\imath}$ be the minimum i such that $|\bigcup_{j=1}^{i} B_j| \geq n/4$. ($\hat{\imath}$ can be found with a prefix-sum computation on the "valid" elements.) Broadcast $\hat{\imath}$.

(8) Mark all "valid" elements in $B_1, \ldots, B_i$ as elements of E.

3.3.2 Second Phase: The *Merging Phase*

Consider the tree R of recursive calls in KLEIN-LIKE-MREDUCE. We observe that, after $l = 3\log_{4/3} p$ levels of R (when the first part of our algorithm stops), the sizes of the ground sets associated with the nodes in R at level l are at most n/p. This is due to the fact that the descendants of a node u in R that are 3 levels below u are *smaller* than u by approximately a factor 3/4. More precisely, if $n(u)$ denotes the size of the ground set of $T(u)$ (the subtree rooted at u) then, for every node w three levels below u, $n(u) \leq 3n(w)/4 + 1$ [13, 14, 15]. Hence, each PQ-tree obtained at the end of the first phase fits completely into the local memory of one processor.

Unfortunately, the same argument does not hold for the reduction sets. Recall that $m = \Sigma_{i=1}^{k}|A_i|$. Let u be an internal node of R, $A_{u_1}, \ldots, A_{u_j}$ its reduction sets, and $m_u = \Sigma_{i=1}^{j}|A_{u_i}|$. Since the sizes of the reduction sets of the children of u depend strictly on the A_{u_i} and on how they intersect with the set E computed for u, it is possible that the A_{u_i} are split in an unbalanced way. That is, we can have $\Sigma_{i=1}^{j}|A_{u_i}|E| = O(m_u)$ and $\Sigma_{i=1}^{j}|A_{u_i}/E| = O(1)$ (or vice versa). If this continues up to level $3\log p$ of R, it is possible that for a recursive call associated with a node v at level l, $\Sigma_{i=1}^{f}|A_v i| > m/p$.

Therefore, while the ground set of $T(v)$, and hence $T(v)$, can fit in one processor, the reduction sets could possibly not. Thus, at this point of the computation, we can not simply use the sequential algorithm of Booth and Lueker [1] for completing the reduction.

Our idea for solving this problem is the following. Let us consider a node v at level l in R that has $m_u > m/p$. Since, at any level of recursion, the sum of the sizes of all reduction sets is at most $2m$, we can create α_v copies of $T(v)$, with $\alpha_v = \lfloor \frac{m_v}{m/p} \rfloor$. We observe that

$$\Sigma_{v\in l}\alpha_v = \Sigma_{v\in l}\lfloor\frac{m_v}{m/p}\rfloor \leq \Sigma_{v\in l}\frac{m_v}{m/p} \leq \frac{p}{m}\Sigma_{v\in l}m_v \leq \frac{p}{m}\cdot 2m = 2p.$$

Hence, we require at most two copies per processor. The reduction problem of each node v at level l of R will be solved by the α_v processors that have copies of $T(v)$. The next step is the *distribution* of the reduction sets associated to v among these α_v processors. Each of these α_v processors can solve locally the problem of reducing $T(v)$ with respect to the reduction sets that it has stored, using Booth and Lueker's algorithm [1]. For each processor, let $T'(v)$ refer to this reduced tree. Now, we need to merge these α_v trees, $T'(v)$. More precisely, we need to compute a PQ-tree $\widehat{T}(v)$ such that $L(\widehat{T}(v)) = L(\overline{T}(v))$, where $\overline{T}(v)$ is the PQ-tree that we would have obtained by reducing $T(v)$ directly with respect its reduction sets. For the construction of $\widehat{T}(v)$, we merge the $T'(v)$ trees in a binary tree fashion as depicted in figure 1.

Algorithm 8 Merging Phase

Input: h PQ-trees $T(i)$, with $|T(i)| \leq n/p$ and $\Sigma_i|T(i)| \leq n$, and their reduction sets.

Output: The $T(i)$ reduced with respect their reduction sets.

(1) Let m_i be the sum of the sizes of the reduction sets of T_i. Make $\alpha_i = \lfloor \frac{m_i}{m/p} \rfloor$ copies of each $T(i)$. Distribute the reduction sets of each T_i between the processors that have the copies of $T(i)$.

(2) Each processor executes the sequential algorithm [1] for its PQ-trees with the reduction sets that it has stored. Let $T'(i)$ refer to the trees obtained.

(3) The α_i processors associated with each $T(i)$ merge the $T'(v)$ trees in a binary tree fashion, as depicted in Figure 1. The details are discussed below.

We will now discuss the details of Step 3. The following Theorem 3 shows that the merge operation in Step 3 of Algorithm 8 reduces to a tree intersection operation [13, 14, 15].

Theorem 3 *Let T be a PQ-tree over the ground set S and let T' be a copy of T. Let T^* and T'^* be the result of the reduction of T with respect to $\{A_1, \ldots, A_r\}$ and of T' with respect to $\{B_1, \ldots, B_t\}$, respectively. Let $\overline{T}$ be the PQ-tree obtained by reducing T with respect to $\{A_1, \ldots, A_r, B_1, \ldots, B_t\}$. Then,*

$$\lambda \in L(\overline{T}) \Leftrightarrow \lambda \in L(T^*) \cap L(T'^*).$$

Proof. $L(\overline{T})$ is the intersection of the sets of all orderings that satisfy $A_1, \ldots, A_r$, $B_1, \ldots, B_t$, and $L(T)$. $L(\overline{T})$ is always the same, independently of the order in which we reduce T. If $\lambda \in L(\overline{T})$, then λ must belong to the intersection between the set of all orderings that satisfy $A_1, \ldots, A_r$ and L(T) and it must also belong to the intersection between the set of all orderings that satisfy $B_1, \ldots, B_t$ and L(T), that is $\lambda \in L(T^*)$, $\lambda \in L(T'^*)$ and $\lambda \in L(T)$. Hence λ belongs to $L(T^*) \cap L(T'^*)$. The reverse can be shown analogously. □

Set E is called *segregated* in T if $E = leaves(lca_T(E))$. If E is contiguous in T, we can modify T, obtaining T', so that E is segregated in T' and $L(T) = L(T')$. Namely, if E is not already segregated, then $v = lca_T(E)$ is a Q-node (from the definition of *contiguous*). Insert an R-node z between the children $v_p, \ldots, v_q$ and v. In the resulting tree T', $z = lcat_{T'}(E)$ and $leaves(z) = \bigcup_{i=p}^{q} leaves(v_i) = E$. Moreover, it follows from the definition of the R-node that $L(T') = L(T)$.

For a given tree T of n nodes, a node v of T with s children determines a separation of T into $s+1$ subtrees: $T_1, \ldots T_s$, the subtree of T rooted at the children of v, and T_0, the subtree obtained from T by deleting $T_1, \ldots, T_s$. We say v is a *good separator* of T if each subtree $T_0, T_1, \ldots, T_s$ has no more than $n/2$ nodes. Every tree with at least 2 nodes has a good separator, that can be found by computing $size(v)$, that is the number of descendants of v, for each node v of T. (This can be easily computed using the Euler tour technique [3].

We recall the following PRAM algorithm from [13, 14, 15]:

Algorithm 9 Intersect(T,T')
Input: 2 PQ-trees T, T' over the same ground set.
(1) If T has only one node, return.
(2) Find a good separator v of T with children $v_1, \ldots, v_s$. Set $A = leaves_T(v)$ and $A_i = leaves_T(v_i)$ for $i = 1, \ldots, s$. Let $T_0 = T/A$ and $T_i = T|A_i$ for $i = 1, \ldots, s$.
(3) Reduce T' with respect to A and then with respect to the disjoint sets $A_1, \ldots, A_s$. If v is a Q-node, also reduce T' with respect to the disjoint sets $A_1 \cup A_2, A_3 \cup A_4, \ldots$ and with respect to the disjoint sets $A_2 \cup A_3, A_4 \cup A_5, \ldots$
(4) Segregate T' with respect to A, and let $T'_0 = T'/A$.
(5) For $i = 1, \ldots, s$, segregate T' with respect to A_i, and let $T'_i = T'|A_i$.
(6) Identify a node v with $lca_{T'}(A)$. The corresponding subtrees T_i and T'_i have identical ground set. Recursively intersect the corresponding subtrees.

We will now outline a coarse grained parallel implementation of Algorithm 9. We observe that, for a coarse grained parallel implementation, the recursion depth reduces to $O(\log p)$ because at that level, subproblems fit into a single processor and can be handled sequentially. Finding a good separator (Step 2) can be done using the Euler tour technique. Since the

two PQ-trees to intersect fit in the local memory of one processor, this computation can be done sequentially in $O(n)$ time and $O(1)$ communication rounds. The same bounds hold for the computation of T_0 and T_i. The reduction of T' required in Step 3 can be done using Booth and Lueker's sequential algorithm. Since T' is on the same ground set as T, $|T'| \leq 2n$. Therefore, we can perform this step in $O(1)$ communication rounds and $O(n)$ time. With a similar argument, segregating T' with respect to A and all the A_i (Step 4), can be performed sequentially in linear time. The same bounds hold for computing T'_0 and all T'_i.

3.3.3 Summary

Theorem 4 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, Algorithm* MReduce *performs a reduce operation for a PQ-tree T in $O(\log^2 p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

3.4 Convex Bipartite Graphs

Recall the definition of convex bipartite graphs (Definition 2). Given a bipartite graph $G = (A, B, E)$ with $A = \{a_1, a_2, \cdots, a_k\}$ and $B = \{b_1, b_2, \cdots, b_n\}$. Let $\mathcal{A} = \{A_1, \ldots, A_k\}$ where $A_i = \{b \in B : (a_i, b) \in E\}$, and let T be a PQ-tree over the ground set B consisting of a root with children $b_1, b_2, \cdots, b_n$. The problem of determining whether G is convex and, if this is the case, computing the correct ordering of the elements in B is equivalent to the *MReduce* operation on T with respect to $\mathcal{A}$.

Theorem 5 *On a coarse grained multicomputer with p processors and $O(\frac{N}{p})$ storage per processor, the problem of determining whether G is convex (and computing the correct ordering of the elements in B) can be solved in $O(\log^2 p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

References

[1] K. S. Booth and G. S. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms," in *Journal of Computer and System Sciences*, Vol. 13, pp. 335-379, 1976.

[2] P. Bose, A. Chan, F. Dehne, and M. Latzel, "Coarse grained parallel maximum matching in convex bipartite graphs," in Proc. *13th International Parallel Processing Symposium* (IPPS'99), 1999.

[3] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song, "Efficient parallel graph algorithms for coarse grained multicomputers and BSP," in Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP'97), Bologna, Italy, 1997, Springer Verlag Lecture Notes in Computer Science, Vol. 1256, pp. 390-400.

[4] E. Caceres, A. Chan, F. Dehne, G. Prencipe, "Coarse grained parallel algorithms for detecting convex bipartite graphs," Technical Report, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6.

[5] L.Chen and Y.Yesha, "Parallel recognition of the consecutive ones property with applications," J. Algorithms, vol. 12, no. 3, pp. 375-392. 1991.

[6] Lin Chen, "Graph isomorphism and identification matrices: parallel algorithms," IEEE Trans. on Parallel and Distr. Systems, Vol. 7, No. 3, March 1996, pp. 308 ff.

[7] F. Dehne (Ed.), "Coarse grained parallel algorithms," Special Issue of *Algorithmica*, Vol. 24, No. 3/4, 1999, pp. 173-426.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers," in Proc. *ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[9] F. Dehne, A. Fabri, and C. Kenyon, "Scalable and Architecture Independent Parallel Geometric Algorithms with High Probability Optimal Time," in Proc. *6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.

[10] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A.A. Kokhar, "A randomized parallel 3D convex hull algorithm for coarse grained multicomputers," in Proc. *ACM Symposium on Parallel Algorithms and Architectures* (SPAA'95), pp. 27–33, 1995.

[11] M.T. Goodrich, "Communication efficient parallel sorting," ACM Symposium on Theory of Computing (STOC), 1996.

[12] X. He and Y.Yeshua, "Parallel recognition and decomposition of two termninal series parallel graphs," Information and Computation, vol. 75, pp. 15-38, 1987

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

[14] P. Klein. "Efficient Parallel Algorithms for Chordal Graphs". *Proc. 29th Symp. Found. of Comp. Sci., FOCS* 1989, pp. 150–161.

[15] P. Klein. "Parallel Algorithms for Chordal Graphs". In *Synthesis of parallel algorithms*, J. H. Reif (editor). Morgan Kaufmann Publishers, 1993, pp. 341–407.

[16] A.C. Tucker, "Matrix characterization of circular-arc graphs," Pacific J. Mathematics, vol. 39,no. 2, pp. 535-545, 1971.

[17] L. Valiant, "A bridging model for parallel computation," Communications of the ACM, Vol. 33, No. 8, August 1990.

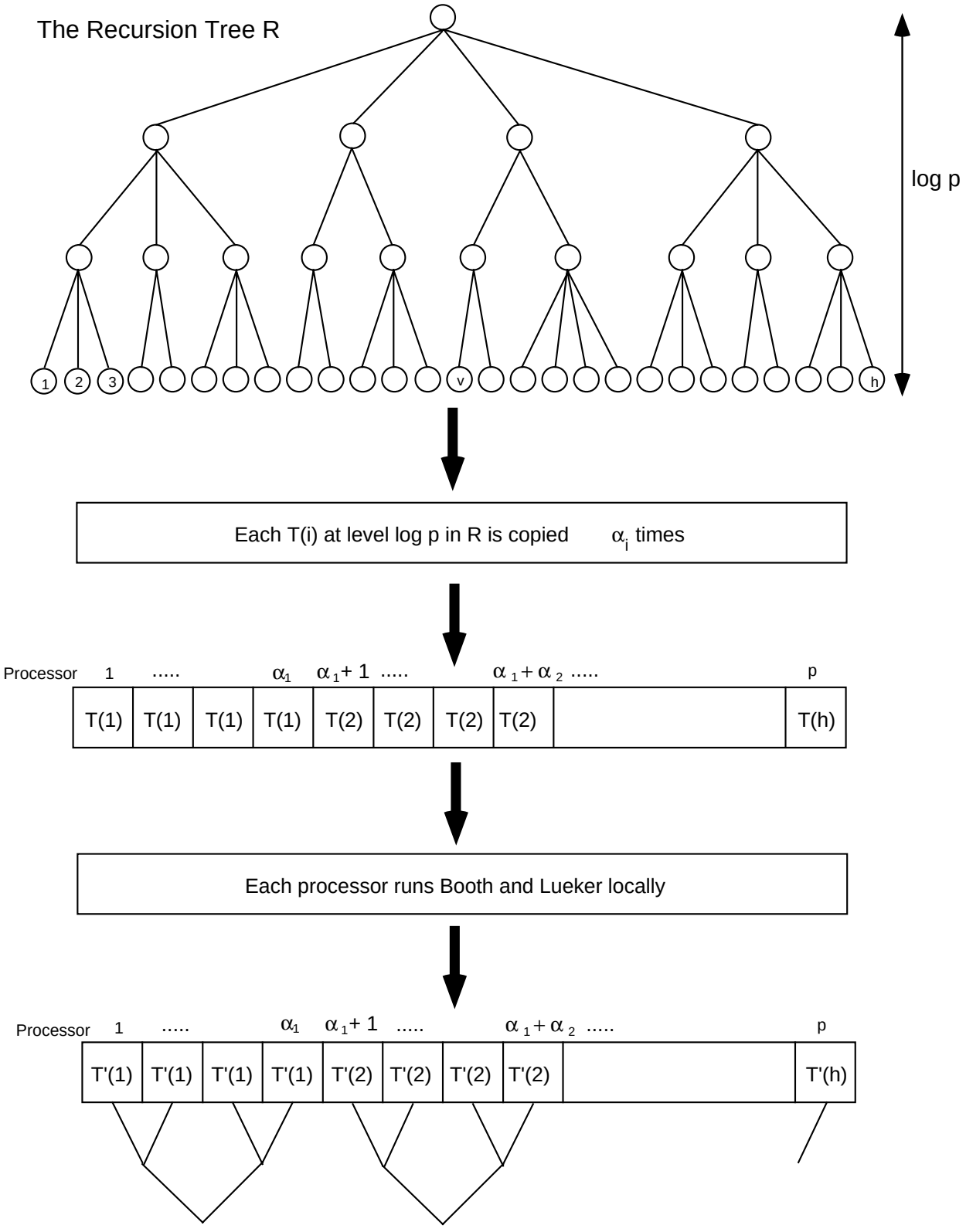

Figure 1: Illustration of the Main Algorithm.