A Practical Parallel Algorithm for Finding the Block Tree of a Simply Connected Graph

Nilabha Dev R.K. Ghosh
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur 208016.
{ndev,rkg}@cse.iitk.ac.in.

March 3, 2001

Abstract

We present a new parallel algorithm for finding the block tree corresponding to the biconnected components of a simply connected graph on the Block Distributed Memory parallel architecture [3]. Our algorithm has a theoretical running time of $\tau \log p + \sigma O(\frac{n^2}{p}) + O(\frac{n^2}{p^2})$. The paper includes an experimental study of the performance of the algorithm with respect to an efficient sequential algorithm.

1 Introduction

The problem of finding the biconnected components of a graph is one of the fundamental problems in algorithmic graph theory. Efficient sequential algorithms for this problem use *depth first search* [6]. However when the graph has a large number of vertices the memory requirement is high and consequently the problem of finding the biconnected components of a graph in the form of a *block tree* is impractical using a sequential algorithm. In this paper a parallel algorithm has been presented to solve the above problem and output the biconnected components in the form of a block tree [5]. An experimental study of the same is also presented.

It is well known that there exists no efficient parallel implementation of depth first search [2]. Hence any parallel implementation must either do depth first search locally and then combine the results on a single processor or else must use non depth first search based techniques.

We assume that the n vertices of a connected graph are evenly partitioned into the p processors so that each processor has $\frac{n}{p}$ vertices. Also this algorithm works only if the graph is connected.

The algorithm given here follows the first approach and is divided into two stages [4]. In the first stage each processor finds the block tree of the sub-graph induced by the subset of vertices local to it. An edge which links a pair of vertices belonging to two block trees across processor boundaries is called a *trans-arc*. A trans-arc is represented as a *block vertex* in each of the two blocktrees which it links. The block vertices representing a trans arc are referred to as *non-local block vertices* in the respective processors to which they belong. All other block vertices are referred to as *local block vertices* in their respective processors. In the next stage the processors merge their respective block graphs forming the block graph of the entire graph. With the right choice of data structure for representing the graphs the running time of the algorithm is found to be $\tau \log p + \sigma O(\frac{n^2}{p}) + O(\frac{n^2}{p^2})$.

The organization of this paper is as follows. The parallel computation model is explained in section 2. Basic graph theoretic definitions and the notions of biconnected components, block tree etc, are introduced in section 3. Section 4 presents the proposed parallel algorithm and includes the running time analysis of the same. The experimental results of the algorithm are discussed in section 5. Finally section 6 presents the conclusions.

2 The Parallel Computation Model

The parallel computation model used in this paper is a collection of powerful general purpose processors connected by a fast interconnection network [3],[1]. It assumes that the memory is distributed and a non local memory access results in interprocessor communication which takes at least an order of magnitude more than local memory access. Also every processor is assumed to have a direct link to every other processor.

Any algorithm for this parallel computational model can be viewed as a series of local computational steps interleaved with communication steps. The time to transfer m consecutive words from a processor to another is given by $\tau + \sigma m$ time where τ is the latency of the network and σ is the time for a processor to inject or receive a single word from the network. The communication time for the algorithm $T_{comm}(n,p)$ is then computed as a function of the input size n and the number of processors p in terms of the parameters τ and σ. $T_{comp}(n,p)$ is defined to be the time for local computation steps. The total running time of the algorithm is then $T_{comp}(n,p) + T_{comm}(n,p)$. The objective is to develop algorithms such that $T_{comp}(n,p) = O(\frac{T_{seq}}{p})$ such that $T_{comm}(n,p)$ is as small as possible, where T_{seq} is the time taken by the best sequential algorithm to solve the given problem.

3 Biconnected Components and Block Tree

Let $G = (V,E)$ be an undirected connected graph with vertex set V and edge set E. Let $\{E_i \mid 1 \leq i \leq k\}$ be a partition of E into a set of k disjoint subsets such that two edges e_1 and e_2 are in the same partition if and only if there is a simple cycle in G containing e_1 and e_2 or $e_1 = e_2$. Let $\{V_i \mid 1 \leq i \leq k\}$ be a collection of sets of vertices where V_i is the set of vertices in E_i for each i, $1 \leq i \leq k$. A vertex v is a *cutpoint* if it appears in more than one vertex set V_i. Subgraph $G_i = (V_i,E_i)$ $\forall$i, $1 \leq i \leq k$ is called a *block* of G. A *block tree* of a connected graph blk(G) is defined as follows. Each *block* and *cutpoint* in the graph is represented as a vertex of blk(G) . The vertices representing the blocks are called *b-vertices* while those representing the cut points are called *c-vertices*. Two vertices u and v of blk(G) are adjacent if and only if u is a *c-vertex*, v is a *b-vertex* and the cutpoint corresponding to u is contained in the block corresponding to v .

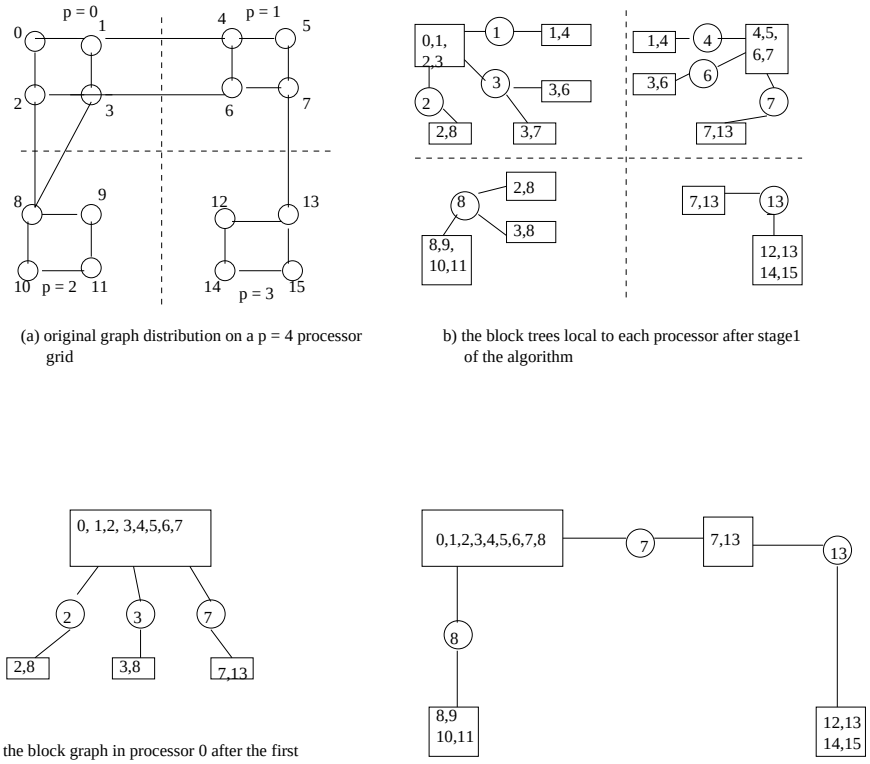
4 The Parallel Algorithm

As mentioned in section 1, initially the vertices of a given graph are assumed to be evenly distributed among the p processors. Each processor is responsible for finding the blocktree of the sub-graph induced by the vertices assigned to it. The algorithm for finding the block tree of a connected graph, presented in this paper assumes that the graph has been evenly distributed on the processors . Each processor has $\frac{n}{p}$ vertices . Also the vertices have been numbered such that processor i has vertices numbered from $\frac{i \cdot n}{p}$ to $\frac{(i+1) \cdot n}{p} - 1$ for $i = 0, 1, 2, \ldots, p\text{-}1$.

The algorithm is divided into two stages. In the first stage all processors locally compute the block tree of the sub-graph induced by the vertices assigned to it using an efficient sequential algorithm. Trans-arcs are represented in their respective components belonging to two different processors. The block trees that do not have at least one b-vertex composed of local vertices are eliminated as they are superfluous. Note that these block trees consist of only non-local vertices. In the second stage the algorithm iteratively merges the block trees of the processors to form the final block tree.

The processors allocate themselves into groups in the second stage. The number of groups decreases by half in each iteration while the number of processors in each group doubles . Initially each group has only two processors. In every iteration the processors elect a group leader, the processor with the lowest index. The processors in each group send their local blocktrees to the group leader, in case there exists an edge in the original graph linking any two sub-graphs assigned to the processors in the group. Eventually the block tree held by each processor is sent to processor 0 as the underlying graph is known to be connected.

There are p = 2^m processors in this computation for some small $m = 1, 2, \ldots \log p$. In iteration i of the algorithm the processors organize themselves into



(a) original graph distribution on a p = 4 processor grid

b) the block trees local to each processor after stage1 of the algorithm

c) the block graph in processor 0 after the first iteration of the second phase. Here block trees of processors 0 adn one are merged in processor 0. In this iteration processors 0 and 1 form one group and processors 2 and 3 form group. No merging takes place in the other group composed of processors 2 and 3 as there is no direct edge between them

d) the final block graph on processor 0 after iteration 2 of the algorithm. In iteration 2 all the 4 processors are in a single group and the edges are merged in the group leader, processor 0.

Figure 1: An example to illustrate the algorithm

groups of 2^{i+1} processors each. Note that in this algorithm the processors do not have to do any more communication steps besides the sending of the sub-graphs because each processor can locally find out the index of the processor to which a vertex j belongs in particular iteration by the following formula.

$$2^i(jdiv(2^i\frac{n}{p})) \tag{1}$$

where j = 0,1,2,...n-1 is the original vertex number at the end of the first stage of the algorithm and i = 0, 1,2,...,log p-1 is the iteration number.

During each iteration a processor can be in one of four different states namely *receiving*, *sending*, *inactive* and *dead*. A processor in *receiving* state receives a sub-graph from a processor in *sending* state. An *inactive* processors does nothing. A processor enters the *dead* state after sending its local sub-graph. *Dead* processors do not change their state in the later iterations and thus the algorithm proceeds to completion.

4.1 The Detailed Parallel Algorithm

The following is run on processor j.

Algorithm 1 *Parallel Algorithm to find the Block Tree of a Connected Graph*
Input
j : my processor number
p : total number of processors
state $\in\{$ *receiving*, *sending*, *inactive* or *dead* $\}$: state of the processor
begin

(i) Processor finds the block tree of the sub-graph induced in it using an efficient sequential algorithm.

(ii) If any block tree is composed of only one c vertex and only one b vertex, that corresponds to a trans arc then eliminate that block tree.

(iii) Initially all processors mark themselves as *sending*.

for $i = 0$ **to** $\log p$ -1

begin

(1) Processor j locates its own group as $g = j$ div $2^{i+1}$and also finds the other processors in its group. The processor calculates its serial number in its group as $l = j \bmod 2^{i+1}$. Besides this the processors find out what are the non-local edges, the vertices corresponding to these non-local edges and the cutpoints in the sub-graph local to each.

(2) If $l = 0$ then processor is a *receiver*.

(3) If processor is a *receiver* then it receives block trees from at most $2^{i+1} - 1$ processors and merges these block trees using an efficient sequential algorithm.

(4) If processor is in *sender* or *inactive* state then it finds the group leader and checks if any of the non-local vertices in it can be found in the group leader in this iteration. If this is not the case then the processor marks itself as *inactive*.

(5) If processor is a *sender* then it sends a block tree to the group leader if any of the non-local vertices in the block tree can be found in the group leader.

(6) If processor is a *sender* and was successful in sending all the block trees in it, it marks itself as *dead*. A *dead* processor does not change its state during the execution of the algorithm.

end

end

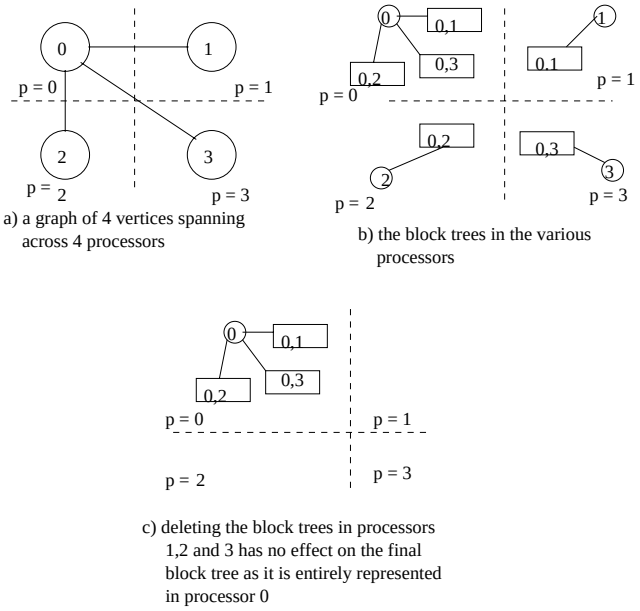


Figure 2: An example to illustrate Lemma 1

4.2 Correctness of the Algorithm

Lemma 1 : Elimination of block trees corresponding to step(ii) of the algorithm does not affect the final computation of the block tree.

Proof : The block tree in Figure 2(b) shows a block tree composed of only two vertices. It consists of a c-vertex and a b-vertex corresponding to a single trans-arc. This block tree can be eliminated, because the b-vertex is also represented in its own component of the processor corresponding to the other end of the trans-arc. That processor can find the complete block tree without referring to this block tree.■

Lemma 2 : Formula (1) correctly calculates the index of the processor in which the non-local vertex of a trans-arc can be found.

Proof : This follows directly from the definition of trans-arc, non-local vertex and Formula (1).■

Lemma 3 : Merging of two block trees gives a new block tree.

Proof : An algorithm for merging the block tree is given as follows

For each edge e in the second blocktree

add e to the the first blocktree using the algorithm given in [5].■

Lemma 4 : The size of the blocktree in iteration i of the algorithm is bounded by $O(2^i \frac{n}{p}) + O(\frac{n^2}{p})$ space requirement.

Proof : So far no mention has been made about the data structure for storing the block graphs up to this stage. While no specific data structure is proposed here, the space requirements of the data structure is linear in the number of edges of the block tree. This is crucial as entire block trees are sent from one processor

to another.

In an iteration i, the total number of local vertices is given by $2^i\frac{n}{p}$. As explained earlier in section 1, a non-local b-vertex contains two block vertices belonging to two different processors. While the local b-vertices are stored explicitly in the block tree representation the non-local b-vertices are represented implicitly in the adjacency information of the respective c-vertices. Therefore non-local b-vertices are not stored locally in a processor.

The space requirement of the local b-vertices is given by the total number of connected components that can be formed from the number of vertices in the processor. The number of vertices local to each processor in iteration i is given by $O(2^i\frac{n}{p})$. The same bound holds for storing the adjacency information of these local b-vertices. The maximum number of c-vertices in the block tree is $2^i\frac{n}{p}$, the space bound being same as that for the b-vertices. For c-vertices that are adjacent to local b-vertices the space requirement is again $O(2^i\frac{n}{p})$ which is the maximum number of local b-vertices in the graph. Now, any c-vertex may be connected to $n - 2^i\frac{n}{p}$ other non-local b-vertices where n is the maximum number of b-vertices and at most $2^i\frac{n}{p}$ b-vertices are local to a processor. Thus the total space requirement for storing the adjacency information is calculated as

$$O(2^i\frac{n}{p}(n - 2^i\frac{n}{p})) \tag{2}$$

This is because in iteration i the maximum number of non-local vertices corresponding to a c-vertex is n - $2^i\frac{n}{p}$. This is in fact the maximum degree of any such c-vertex. For $i = 0$ (i.e. the first iteration) the space requirement is maximum and is given by $O(\frac{n^2}{p})$ thus giving the total space requirement as $O(2^i\frac{n}{p})$ + $O(\frac{n^2}{p})$.■

Theorem 1 : The above algorithm correctly calculates the block tree of the algorithm in time $\tau\log p + \sigma O(\frac{n^2}{p}) + O(\frac{n^2}{p^2})$.

Proof : The algorithm terminates because a processor after sending the block tree to its group leader marks itself as *dead*. At the end of $\log p$ steps of the algorithm, all the processors except 0 are marked *dead*.

In the first stage of the algorithm the running time for calculating the local block tree is given as $O(\frac{n^2}{p^2})$ + $O(\frac{n}{p})$. This is because the maximum number of edges in a graph of n vertices is given by $\binom{n}{2}$ which is $O(n^2)$.

In the second stage the running time of the algorithm is divided into two parts. T_{comp} and T_{comm}.

In iteration i, the running time is given by

$T(i) = T_{comp}(i) + T_{comm}(i)$.

Now $T_{comm}(i) = \tau + \sigma S(i)$. where $S(i)$ is the maximum size of the block tree in iteration i.

The computation time for the algorithm is given by the time required by the group leader to merge the sub-graphs in iteration i. Therefore $T_{comp}(i)$=$N(i).S(i)$

where N(i) is the number of sub-graphs in i^{th} iteration that are being merged and S(i) is as defined earlier. The number N(i) is equal to 2^i. The merging step is linear in the number of edges of blocktrees being merged [5].

Therefore the total time for the second stage is given by

$$\begin{aligned} \mathrm{T} &= \sum_{i=0}^{\log p-1} T_{comm}(i) + \sum_{i=0}^{\log p-1} T_{comp}(i) \\ &= \sum_{i=0}^{\log p-1} (\tau + \sigma S(i)) + \sum_{i=0}^{\log p-1} N(i).S(i). \\ &= O(\tau \log p) + \sigma \sum_{i=0}^{\log p-1} S(i) + \sum_{i=0}^{\log p-1} N(i)S(i) \end{aligned}$$

Now using Equation 2 of Lemma 4, we obtain the time bound as
= O(τlog p + σO($\frac{n^2}{p}$)+ O(np).

Therefore the total time bound for the algorithm is given as τlog$p$ + σO($\frac{n^2}{p}$) + O($\frac{n^2}{p^2}$) for $n > p^3$.■

5 Experimental Results

The parallel blocktree finding algorithm presented here was implemented using MPI/C on a 16 node IBM SP machine. A sequential algorithm was also implemented for purposes of comparison with the parallel algorithm. This algorithm is detailed in [5]. The maximum size of the graph taken was one of 2000 vertices. The algorithm was tested for 4, 8 and 16 processors. A speed-up of a factor of 5 to 8 is obtained using 16 processors over the sequential algorithm. The results were in line with theoretical calculations. However the proposed theoretical speedup of p was not obtained. This is because of the latent communication cost, given as τ in the calculation .

In the graph shown in Figure 3 the time in milliseconds is represented on the y-axis in a log scale and the number of vertices is represented on the x-axis in a linear scale. The above graph shows that for larger values of p, the results are much better. For graphs with large number of vertices the number of MPI messages may increase substantially. Hence there may be a increase in overhead for exchanging messages. However the trend of plots indicate that the parallel algorithm presented in this paper is expected to perform very well in practice for large graphs.

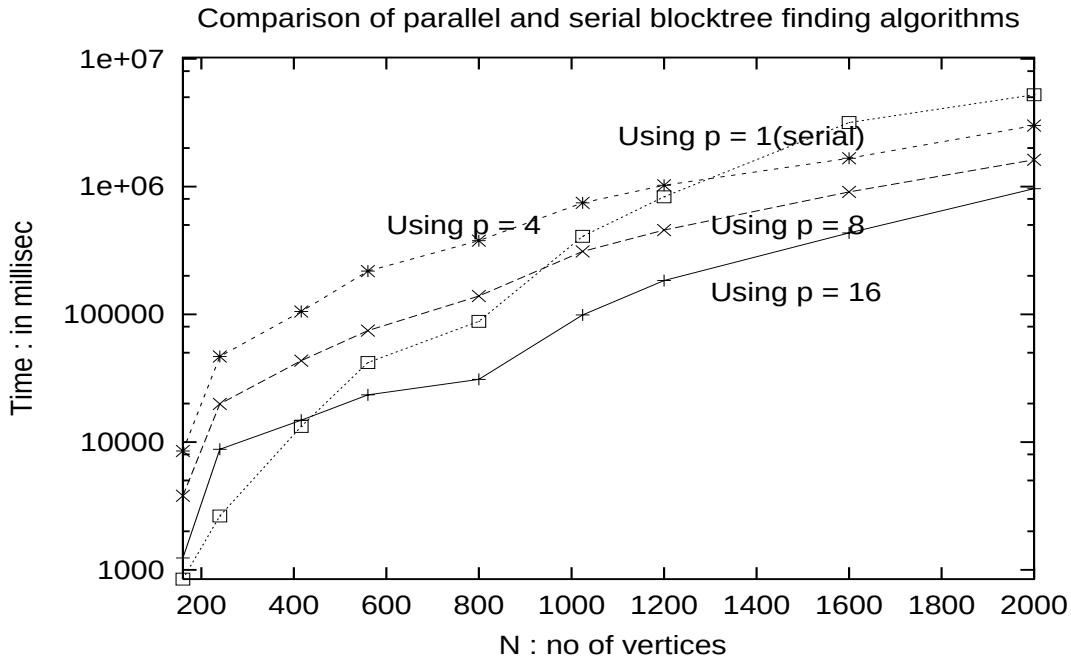


Figure 3: The experimental results

6 Conclusion

In this paper a practical parallel algorithm for finding the biconnected components of graph in the form of a block tree was developed. It was shown through experimental evidences that the performance of the algorithm is indeed close to theoretical results. A huge number of important scientific and computer science problems can be conveniently modeled as graphs. For problems of huge sizes using a sequential algorithm to solve the problem is not always feasible. Hence, there is a need to find efficient parallel algorithms to solve these kind of problems.

References

[1] D. A. Bader, D. R. Helman, and J. F. JaJa, *Practical parallel algorithms for personalized communication and integer sorting*, ACM Journal of Experimental Algorithmics, 1 (1996), pp. 1–42.

[2] J. F. JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley, USA, 1992.

[3] J. F. JaJa and K. W. Ryu, *The block distributed memory model*, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), pp. 830–840.

[4] G. Karypis and V. Kumar, *Parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 71–95.

[5] A. Rosenthal and A. Goldner, *Smallest augmentations to biconnect a graph*, SIAM Journal of Computing, 6 (1977).

[6] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM Journal of Computing, 1 (1972).