COMPUTING BICONNECTED COMPONENTS ON A HYPERCUBE*

Jinwoon Woo and Sartaj Sahni

University of Minnesota

Abstract

We describe two hypercube algorithms to find the biconnected components (i.e., blocks) of a connected undirected graph. One is a modified version of the Tarjan-Vishkin algorithm. The two hypercube algorithms were experimentally evaluated on an NCUBE/7 MIMD hypercube computer. The two algorithms have comparable performance and efficiencies as high as 0.7 were observed.

Keywords and phrases

Hypercube computing, MIMD computer, parallel programming, biconnected components

* This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

1. INTRODUCTION

In this paper we develop two biconnected component (i.e., block) algorithms suitable for medium grained MIMD hypercubes. The first algorithm is an adaptation of the algorithm of Tarjan and Vishkin [TARJ85]. Tarjan and Vishkin provide parallel CRCW and CREW PRAM implementations of their algorithm. The CRCW PRAM implementation of Tarjan and Vishkin runs in $O(\log n)$ time and uses $O(n+m)$ processors. Here n and m are, respectively, the number of vertices and edges in the input connected graph. The CREW PRAM implementation runs in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors.

A PRAM algorithm that use p processors and t time can be simulated by a p processor hypercube in $O(t\log^2 p)$ time using the random access read and write algorithms of Nassimi and Sahni [NASS81]. The CREW PRAM algorithm of [TARJ85] therefore results in an $O(\log^3 n)$ time $O(n+m)$ processor hypercube algorithm. The CREW PRAM algorithm results in an $O(\log^4 n)$ time $O(n^2/\log^2 n)$ processor hypercube algorithm. Using the results of Dekel, Nassimi, and Sahni [DEKE81] the biconnected components can be found in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors.

The processor-time product of a parallel algorithm is a measure of the total amount of work done by the algorithm. For the three hypercube algorithms just mentioned, the processor-time product is, respectively, $O(n^2\log^3 n)$ (assuming $m \approx O(n^2)$), $O(n^2\log^2 n)$, and $O(n^3\log n)$. In each case the processor-time product is larger than that for the single processor biconnected components algorithm ($O(n^2)$ when $m = O(n^2)$). As a result of this, we do not expect any of the above three hypercube algorithms to outperform the single processor algorithm unless the number of available processors, p, is sufficiently large. For example, if $n = 1024$, then the CRCW simulation on a hypercube does $O(\log^3 n) \approx 1000$ times more work than the uniprocessor algorithm. So we will need approximately 1000 processors just to break even.

In fact, the processor-time product for many of the asymptotically fastest parallel hypercube algorithms exceeds that of the fastest uniprocessor algorithm by at least a multiplicative factor of $\log^k n$ for some k, $k \geq 1$. As a result, the simulation of these algorithms on commercially available hypercubes with a limited number of processors does not yield good results. Consequently, there is often a wide disparity between the asymptotic algorithms developed for PRAMs and fine grained distributed memory parallel computers (eg. SIMD and MIMD hypercubes) and those developed for commercially available parallel computers. For example, Ranka and Sahni ([RANK88]) develop an asymptotically optimal algorithm for image template matching on a fine grained MIMD hypercube that has as many processors as pixels in the image. In the same paper, they develop a totally different algorithm for the NCUBE MIMD hypercube. This latter algorithm is observed to be exceedingly efficient and the authors point out that they do not expect similar results by adapting their optimal fine grain algorithm. [RANK89] and [WOO88] are two other examples of this phenomena. The Hough transform algorithm actually used on the NCUBE in [RANK89] bears no resemblance to the asymptotically efficient algorithms developed in the same paper and the practically efficient

connected components algorithm for the NCUBE hypercube developed in [WOO88] is far removed from the asymptotically fast algorithm of [SHIL82].

While in the examples cited above the algorithms that give good performance on a real hypercube are very different from those developed for PRAMs and even those developed specifically for hypercubes whose size is matched to the problem size, in the case of the biconnected components problem we can get good performance by using some, though not all, of the ideas behind the asymptotically best PRAM algorithm. By careful adaptation of the Tarjan and Vishkin algorithm ([TARJ85]) we are able to obtain a good NCUBE algorithm.

The second biconnected components algorithm we consider differs from the Tarjan and Vishkin algorithm in certain crucial steps. Its correctness follows from that of the uniprocessor biconnected components algorithm of [READ68].

The remainder of this paper is organized as follows. In Section 2 we introduce the performance measures we shall be using. Then in Section 3 we describe the algorithm of Tarjan and Vishkin [TARJ85] and also our adaptation to the NCUBE hypercube. Read's uniprocessor biconnected components algorithm [READ68] and our second NCUBE biconnected components algorithm are described in Section 4. The results of our experiments conducted on a 64 processor NCUBE/7 hypercube are described in Section 5.

2. PERFORMANCE MEASURES

The performance of uniprocessor algorithms and programs is typically measured by their time and space requirements. For multicomputers, these measures are also used. We shall use t_p and s_p to, respectively, denote the time and space required on a p node multicomputer. While s_p will normally be the total amount of memory required by a p node multicomputer, for distributed memory multicomputers it is often more meaningful to measure the maximum local memory requirement of any node. This is so as, typically, such multicomputers have equal size local memory on each node.

To determine the effectiveness with which the multicomputer nodes are being used, one also measures the quantities *speedup* and *efficiency*. Let t_0 be the time required to solve the given problem on a single node using the conventional uniprocessor algorithm. Then, the *speedup*, S_p, using p processors is:

$$S_p = \frac{t_0}{t_p}$$

Note that t_1 may be different from t_0 as in arriving at our parallel algorithm, we may not start with the conventional uniprocessor algorithm.

The *efficiency*, E_p, with which the processors are utilized is:

$$E_p = \frac{S_p}{p}$$

Barring any anomalous behavior as reported in [LAI84], [LI86], [QUIN86], and [KUMA88], the speedup will be between 0 and p and the efficiency between 0 and 1.

While measured speedup and efficiency are useful quantities, neither give us any information on the scalability of our parallel algorithm to the case when the number of processors/nodes is increased from that currently available. It is clear that, for any fixed problem size, efficiency will decline as the number of nodes increases beyond a certain threshold. This is due to the unavailability of enough work, i.e., processor starvation. In order to use increasing numbers of processors efficiently, it is necessary for the work load (i.e, t_0) and hence problem size to increase also [GUST88]. An interesting property of a parallel algorithm is the amount by which the work load or problem size must increase as the number of processors increases in order to maintain a certain efficiency or speedup. Kumar, Rao, and Ramesh [KUMA88] have introduced the concept of isoefficiency to measure this property. The *isoefficiency*, $ie(p)$, of a parallel algorithm/program is the amount by which the work load must increase to maintain a certain efficiency.

We illustrate these terms using matrix multiplication as an example. Suppose that two $n \times n$ matrices are to be multiplied. The problem size is n. Assume that the conventional way to perform this product is by using the classical matrix multiplication algorithm of complexity $O(n^3)$. Then, $t_0 = cn^3$ and the work load is cn^3. Assume further that p divides n. Since the work load is easily evenly distributed over the p processors,

$$t_p = \frac{t_0}{p} + t_{com}$$

where t_{com} represents the time spent in interprocessor communication.
So, $S_p = t_0/t_p = pt_0/(t_0+pt_{com})$ and $E_p = S_p/p = t_0/(t_0+pt_{com}) = 1/(1+pt_{com}/t_0)$. In order for E_p to be a constant, pt_{com}/t_0 must be equal to some constant $1/\alpha$. So, $t_0 =$ work load $= cn^3 = \alpha pt_{com}$. In other words, the work load must increase at least at the rate αpt_{com} to prevent a decline in efficiency. If t_{com} is ap (a is a constant), then the work load must increase at a quadratic rate. To get a quadratic increase in the work load, the problem size n needs increase only at the rate $p^{2/3}$ (or more accurately, $(a\alpha/c)^{1/3}p^{2/3}$).

Barring any anomalous behavior, the work load t_0 for an arbitrary problem must increase at least linearly in p as otherwise processor starvation will occur for large p and efficiency will decline. Hence, in the absence of anomalous behavior, $ie(p)$ is $\Omega(p)$. Parallel algorithms with smaller $ie(p)$ are more scalable than those with larger $ie(p)$.

The concept of isoefficiency is useful because it allows one to test parallel programs using a small number of processors and then predict the performance for a larger number of processors. Thus it is possible to develop parallel programs on small hypercubes and also do a performance evaluation using smaller problem instances than the production instances to be solved when the program is released for commercial use. From this performance analysis and the isoefficiency analysis one can obtain a reasonably good estimate of the program's performance in the target commercial environment where the multicomputer may have many more

processors and the problem instances may be much larger. So with this technique we can eliminate (or at least predict) the often reported observation that while a particular parallel program performed well on a small multicomputer it was found to perform poorly when ported to a large multicomputer.

In order to achieve good efficiency it is necessary to use a parallel algorithm that does a total amount of work comparable to that done by the fastest uniprocessor algorithm. The efficiency of the three hypercube algorithms described above decreases to zero as n increases.

3. THE TARJAN-VISHKIN BICONNECTED COMPONENTS ALGORITHM

The fastest uniprocessor algorithm for biconnected components uses depth first search ([TARJ72], [HORO78]). Currently there are no efficient parallel algorithms to perform depth first search on general graphs. The parallel biconnected components algorithm of [TARJ85] uses a different uniprocessor algorithm as its starting point. This is reproduced in Figure 1. The strategy to find the biconnected components of the connected graph G is to first construct an auxiliary graph G' such that the vertices of G' correspond to the edges of G. Further the connected components of G' correspond to the biconnected components of G. I.e., two vertices of G' are in the same connected component of G' iff the corresponding edges of G are in the same biconnected component of G.

In the CRCW implementation of Figure 1 described in [TARJ85], the spanning tree T of Step 1 is found by using a modified version of Shiloach and Vishkin's connected components algorithm [SHIL82]. The preorder number and number of descendants is found using a doubling technique ([WYLL79], [NASS80]) and an Eulerian tour. Step 2 is done using the doubling technique. Step 3 is straightforward. The components of G'' are found using the algorithm of [SHIL82]. Step 5 is straightforward.

In our hypercube implementation of Figure 1, we begin with an adjacency matrix representation of the connected graph G. This is partitioned over the available p processors using the balanced scheme of [WOO88]. The spanning tree of Step 1 is found using a modified version of the hypercube connected components algorithm of [WOO88]. This performs better than the hypercube adaptation of the algorithm of [SHIL82]. While the preorder number and number of descendants can be found in $O(\log^2 n)$ time on an n node hypercube using the steps outlined in [GOPA85], we did not attempt to map this $O(\log^2 n)$ algorithm onto a p node hypercube. When $p \ll n$ it is more efficient to broadcast the spanning tree to all hypercube nodes and let each compute the preorder number and number of descendants for all tree vertices using the traditional uniprocessor algorithm [HORO86]. This is even faster than having one processor compute the preorder number and number of descendants of each node and then broadcast the results to the remaining processors. This is so as using either approach it is necessary to broadcast the spanning tree as it is needed for future steps. When the latter approach is used the preorder number and number of descendants of each tree vertex also

Step 1 Find a spanning tree T of G using any linear-time search method. Number the vertices of G from 1 to n in preorder and identify each vertex by its preorder number. Let $i \rightarrow j$ denote an edge in T such that i is the parent of j. Compute the number of descendants $nd(v)$ of each vertex v by processing the vertices in postorder using the recurrence $nd(v)=1+\Sigma\{nd(w) | v \rightarrow w \text{ in } T\}$. (We regard every vertex as a descendant of itself.) A vertex w is a descendant of another vertex v if and only if $v \leq w \leq v+nd(v)-1$.

Step 2 For each vertex v, compute $low(v)$, the lowest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G-T$, and $high(v)$, the highest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G-T$. The complete set of $2n$ *low* and *high* vertices can be computed in $O(n+m)$ time by processing the vertices of T in postorder using the following recurrences ((v,w) denotes an undirected edge):
$low(v) = \min(\{v\} \cup \{low(w) | v \rightarrow w \text{ in } T\} \cup \{w | (v,w) \text{ in } G-T\})$;
$high(v) = \max(\{v\} \cup \{high(w) | v \rightarrow w \text{ in } T\} \cup \{w | (v,w) \text{ in } G-T\})$.

Step 3 Construct G'', the subgraph of G' induced by the edges of T, as follows. For each edge (w,v) in $G-T$ such that $v+nd(v) \leq w$, add $((p(v),v), (p(w),w))$ to G''. For each edge $v \rightarrow w$ of T such that $v \neq 1$ add $((p(v),v), (v,w))$ to G'' if $low(w) < v$ or $high(w) \geq v+nd(v)$.

Step 4 Find the connected components of G'' using any kind of linear-time search.

Step 5 Extend the equivalence relation on the edges of T (the vertices of G'') to the edges of $G-T$ by defining (v,w) equivalent to $(p(w),w)$ for each edge (v,w) of $G-T$ such that $v < w$.

Figure 1: Biconnected component algorithm
Reproduced from [TARJ85].

have to be broadcast.

For Step 2, each hypercube node computes a tentative low and high value for each vertex based on the edge information in its adjacency matrix partition. Then the hypercube nodes communicate with one another to determine the true low and high values. Finally, these values are broadcast to all processors.

Steps 3 and 4 of Figure 1 partition the edges of the spanning tree into equivalence

classes. Each equivalence class gives the vertices of G'' that are in the same component. In our implementation, each node identifies some of the edges of G'' using only the information in its adjacency matrix partition. As each edge is identified it is passed along to an equivalence class construction algorithm based on the union-find algorithms of [HORO86]. At the end each hypercube node has partitioned the tree edges into equivalence classes based on some subgraph of G''. The equivalence classes of all nodes are then merged together using a binary tree merge scheme as in [WOO88]. Following this we have the desired set of equivalence classes.

The final equivalence classes are then broadcast to all nodes. Each node uses this information to classify the non spanning tree edges .

The hypercube implementation is summarized in Figure 2 and an example provided in Figure 3.

Analysis of Figure 2

As it is customary to exclude the data loading (Step 0) time from the run time analysis of parallel algorithms, we shall only consider the time needed to perform Steps 1 through 5. To simplify the analysis, we assume that the union-find algorithms used in Steps 3 and 4 and also in Step 1 [WOO88] run in linear time (actually, the run time is slightly more than linear [HORO86]). Further, let t_i and t_c be constants that reflect the cost of cpu operations and communication. Let α be the time needed to set up an interprocessor communication. So, it takes $\alpha+bt_c$ time to transmit b bytes between adjacent hypercube nodes. For simplicity, constant factors relating the relative complexity of various algorithms are omitted. With these assumptions we obtain the following:

Step 1 a) $(n^2/p + n\log p)t_i + n\log p\ t_c + \alpha\log p$ [WOO88]
b) $n\log p\ t_c + \alpha\log p$
c) nt_i

Step 2

a) $(n^2/p + n)t_i$ (each node has $n(n-1)/2p$ bits of the adjacency matrix; for simplicity we approximate this to n^2/p; the n term represents the $n-1$ edges of the spanning tree.)

b) $n\log p(t_i + t_c) + \alpha\log p$

c) $n\log p\ t_c + \alpha\log p$

Step 3&4 a) $n^2/p\ t_i$
b) $n\log p\ t_i + n\log p\ t_c + \alpha\log p$
c) $n\log p\ t_c + \alpha\log p$

Step 5 $n^2/p\ t_i$

Step 0 Distribute the adjacency matrix to the p hypercube nodes using the balanced method of [WOO88].

Step 1

a) Obtain a spanning tree, in node 0, using a modified version of the balanced connected component algorithm of [WOO88].

b) Broadcast the spanning tree to all hypercube nodes.

c) Each hypercube node traverses its copy of the spanning to determine the preorder number and number of descendants of each tree vertex.

Step 2

a) Each hypercube node computes $low(v)$ and $high(v)$ for each vertex v. This is done using only information available in the node.

b) A binary tree scheme is used to combine the low and high values computed in a) so as to obtain the true low and high values in node 0.

c) Node 0 broadcasts the true low and high values to all hypercube nodes.

Step 3&4

a) Each hypercube nodes (implicitly) constructs a subgraph of G'' using only information available to it. As each edge of this subgraph is identified it is immediately consumed by a union-find type equivalence class algorithm.

b) The equivalence classes of a) are merged using a binary tree merge scheme as in [WOO88].

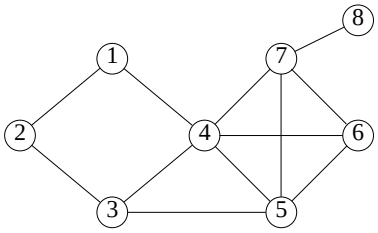
c) The final equivalence classes are broadcast by node 0 to all hypercube nodes.

Step 5 Each hypercube node now classifies each of the non spanning tree edges in its partition into one of the equivalence classes constructed.

Figure 2: Hypercube adaptation of the Tarjan-Vishkin biconnected components algorithm

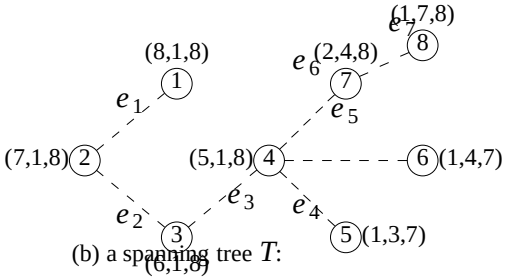
---

Adding the step times together we get the run time, t_{TV}, of our hypercube adaptation of the Tarjan-Vishkin algorithm:



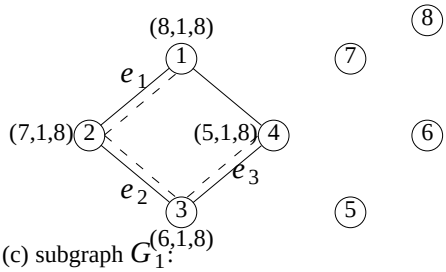
(a) initial graph $G = G_1 \cup G_2 \cup G_3$

G_i's in (c),(d), and (e) below include only dark edges.



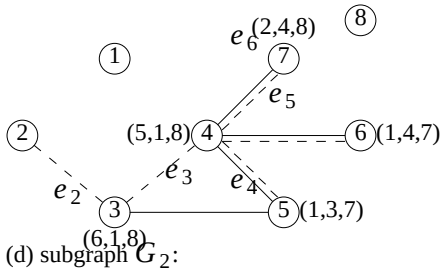
(b) a spanning tree T:

The vertex numbers are arranged in preorder in T and vertex 1 is the root of T. The numbers in parenthesis represent the number of descendants, low, and high, respectively after steps 1 and 2.



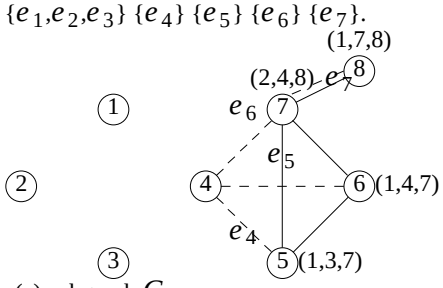
(c) subgraph G_1:

Edge (2,3) $\in T$: $\{e_1,e_2\}$ since $low(3) < 2$.
Edge (3,4) $\in T$: $\{e_2,e_3\}$ since $low(4) < 3$.
Resulting equivalence classes are
$\{e_1,e_2,e_3\}$ $\{e_4\}$ $\{e_5\}$ $\{e_6\}$ $\{e_7\}$.



(d) subgraph G_2:

Edge (4,5) $\in T$: $\{e_3,e_4\}$ since $low(5) < 4$.
Resulting equivalence classes are
$\{e_1\}$ $\{e_2\}$ $\{e_3,e_4\}$ $\{e_5\}$ $\{e_6\}$ $\{e_7\}$.



(e) subgraph G_3:

Edge (5,6) $\in G-T$: $\{e_4,e_5\}$ since $5 + nd(5) \le 6$.
Edge (5,7) $\in G-T$: $\{e_4,e_6\}$ since $5 + nd(5) \le 7$.
Edge (6,7) $\in G-T$: $\{e_5,e_6\}$ since $6 + nd(6) \le 7$.
Resulting equivalence classes are
$\{e_1\}$ $\{e_2\}$ $\{e_3\}$ $\{e_4,e_5,e_6\}$ $\{e_7\}$.

Merging equivalence classes of G_1,G_2,G_3 in step 4 results in $\{e_1,e_2,e_3,e_4,e_5,e_6\}$ $\{e_7\}$.

Figure 3

$$t_{TV} = (4n^2/p + 3n\log p + 2n)t_i + 6n\log p\ t_c + 6\alpha\log p$$

$$= O((n^2/p + n\log p)t_i + n\log p\ t_c + \alpha\log p)$$

The speedup, S_p^{TV}, is therefore:

$$S_p^{TV} = \frac{n^2 t_i}{t_{TV}}$$

and the efficiency, E_p^{TV}, is

$$E_p^{TV} = S_p^{TV}/p$$

$$= \frac{n^2 t_i/p}{t_{TV}}$$

$$= O\left(\frac{1}{1 + \frac{p\log p}{n} + \frac{p\log p}{n}\frac{t_c}{t_i} + \frac{p\log p}{n^2}\frac{\alpha}{t_i}}\right)$$

For constant efficiency, we require

$$\frac{p\log p}{n}\left(1 + \frac{t_c}{t_i} + \frac{1}{n}\frac{\alpha}{t_i}\right)$$

to be constant. For large n this is achieved when n grows at the rate $p\log p$. Or the workload, n^2, grows at the rate $p^2\log^2 p$. The isoefficiency is $O(p^2\log^2 p)$. The scalability of the algorithm is very good as the problem size n needs to grow only at the rate $\Omega(p\log p)$ to ensure that the efficiency does not decline. Suppose we obtain a certain efficiency for graphs with $n = n_1$ and p processors. To obtain the same efficiency with $2p$ processors we need to be using graphs with $n \approx 2n_1(\log 2p/\log p) = 2n_1((1+\log p)/\log p) = 2n_1(1+1/\log p)$. So, if $p = 2$ then n must be approximately $4n_1$; If $p = 4$, n must be approximately $3n_1$; and if $p = 32$, n must be approximately $2.4n_1$. >From S_p^{TV} and t_{TV} we see that good speedups can be expected when $n^2/p \gg n\log p$ and $n^2/p\ t_i \gg n\log p\ t_c + \alpha\log p$.

4. READ'S ALGORITHM AND SECOND BICONNECTED COMPONENTS ALGORITHM

Read [READ68] proposed the algorithm of Figure 4 to find the biconnected components of a connected graph. It is clear that each set S_i at the start of Step 3 contains only edges that are in the same biconnected component. If two sets S_i and S_j have a common edge, then $S_i \cup S_j$ defines an edge set that is in the same biconnected component. Note also that if S_i and S_j have a common edge, this edge must be a spanning tree edge. The correctness of Read's algorithm is established in [READ68].

As stated in Figure 4, Read's algorithm has a complexity $\Omega(ne)$ as each S_i may contain $O(n)$ edges and the number of S_i's is $e-n+1$. The algorithm of Figure 4 can be modified to work with biconnected components of subgraphs of the original graph rather than with the fundamental cycles. The resulting modification is given in Figure 5. An example to illustrate

this algorithm is given in Figure 6. Assuming an adjacency matrix representation, Steps 1 and 2 take $O(n^2)$ time, Step 3 takes $O(n + n^2/p)$ for each subgraph or $O(n^2 + np)$ for all p subgraphs. Each merge of Step 4 takes $O(n)$ (actually it is slightly higher; but in keeping with the simplifying assumption made in Section 3 we assume a linear time complexity for the union-find algorithms). A total of $p-1$ merges are performed. The total complexity of Step 4 is $O(np)$. Step 5 takes $O(n^2)$ time. The overall time is $O(n^2 + np)$. For $p = O(n)$ this is the same asymptotic complexity as for the depth first search algorithm and that of Figure 1 beginning with an adjacency matrix.

Step 1 Find a spanning tree of the graph.

Step 2 Use this spanning tree to obtain a fundamental cycle set for the graph.

Step 3 Let S_i be the set of edges on the fundamental cycle C_i. Repeatedly merge together pairs (S_i,S_j) such that S_i and S_j have a common edge. Each of the edge sets that remain defines a biconnected component of the original graph.

Figure 4: Read's biconnected components algorithm

The p processor hypercube version of Figure 5 takes the form given in Figure 7.

Analysis of Figure 7

We make the same simplifying assumptions as in the case of Figure 2.

Step 1 a) $(n^2/p + n\log p)t_i + n\log p\ t_c + \alpha\log p$ [WOO88]
b) $n\log p\ t_c + \alpha\log p$

Step 2 $(n + n^2/p)t_i$

Step 3 a) $n\log p\ t_i + n\log p\ t_c + \alpha\log p$
b) $n\log p\ t_c + \alpha\log p$

Step 4 $n^2/p\ t_i$

Adding these we obtain

$$t_R = (3n^2/p + n\log p + n)t_i + 4n\log p\ t_c + 4\alpha\log p$$

$$= O((n^2/p + n\log p)t_i + n\log p\ t_c + \alpha\log p)$$

Comparing t_R with t_{TV} we see that the adaptation of Tarjan and Vishkin (Figure 2)

Step 1	Find a spanning tree of the graph.
Step 2	Arbitrarily partition the edges of the graph to obtain p subgraphs $G_1,...,G_p$.
Step 3	Find the biconnected components of each of these subgraphs. Only the spanning tree edges in the components are retained.
Step 4	Let S_i be the edge set in the i'th biconnected component. Merge together the S_i's as in Step 3 of Figure 4.
Step 5	Add the non tree edges to the biconnected components.

Figure 5: Subgraph modification of Figure 4

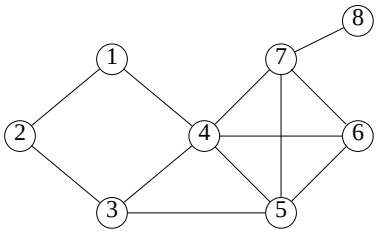
requires 50% more interprocessor communication than does our algorithm of Figure 7. However since the computation time is $O(n^2/p+n\log p)$ vs $O(n\log p\ t_c+\alpha\log p)$ for the communication time, the communication time is an important factor only for small n. So, for small n we expect the algorithm of Figure 7 to outperform that of Figure 2 because of the communication requirements. For any number of processors p, as n is increased the effect of the communication time reduces. At some threshold value the relative performance of the two algorithms is determined by their relative computation time. Because of the simplifying assumptions made in the analysis, the constants in t_{TV} and t_R don't give any indication of this and we need to rely on experiment.

On the other hand if we hold n fixed and increase p the effect of the communication time becomes more significant and the algorithm of Figure 7 can be expected to outperform that of Figure 2.

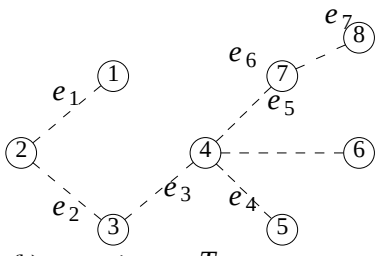
The "big oh" t_R and t_{TV} (obtained by dropping constant coefficients and low order terms) are the same. Hence the "big oh" ${S_p}^R$, ${E_p}^R$, are the same as for the Tarjan-Vishkin adaptation. The isoefficiency for Figure 7 is, therefore, also the same ($\Omega(p^2\log^2 p)$) as for the Tarjan-Vishkin adaptation.

5. EXPERIMENTAL RESULTS

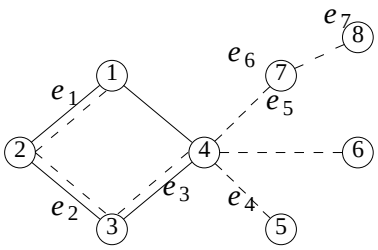
The hypercube algorithms of Sections 3 and 4 were programmed in FORTRAN and run on an NCUBE hypercube multicomputer. In both cases, the last step (i.e., the one to extend the equivalence classes to the non spanning tree edges) was excluded because of memory limitations in the hypercube node processors. For each n, 30 random graphs with edge density



(a) initial graph $G = G_1 \cup G_2 \cup G_3$

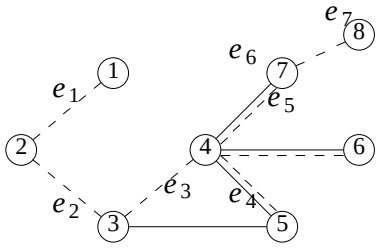


(b) a spanning tree T



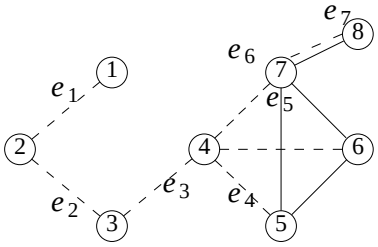
(c) subgraph $G_1 \cup T$:

Resulting equivalence classes are
$\{e_1,e_2,e_3\}$ $\{e_4\}$ $\{e_5\}$ $\{e_6\}$ $\{e_7\}$.



(d) subgraph $G_2 \cup T$:

Resulting equivalence classes are
$\{e_1\}$ $\{e_2\}$ $\{e_3,e_4\}$ $\{e_5\}$ $\{e_6\}$ $\{e_7\}$.



(e) subgraph $G_3 \cup T$:

Resulting equivalence classes are
$\{e_1\}$ $\{e_2\}$ $\{e_3\}$ $\{e_4,e_5,e_6\}$ $\{e_7\}$.

Merging equivalence classes of G_1,G_2,G_3 in step 4 results in $\{e_1,e_2,e_3,e_4,e_5,e_6\}$ $\{e_7\}$.

Figure 6

ranging

from 70% to 90% were generated. The average efficiency is given in the tables of Figure 8 (hypercube implementation of Tarjan-Vishkin algorithm) and Figure 9 (hypercube adaptation of Read's algorithm).

The speedups obtained by the two algorithms for n = 256, 512, and 1024 are plotted in Figures 10, 11, and 12 respectively. In computing the speedups and efficiencies we used the uniprocessor depth first search algorithm of [TARJ72] to obtain t_0. Since the measured

Step 0 Distribute the adjacency matrix to the p hypercube nodes using the balanced method of [WOO88].

Step 1

a) Obtain a spanning tree, in node 0, using a modified version of the balanced connected component algorithm of [WOO88].

b) Broadcast the spanning tree to all hypercube nodes.

Step 2 Each hypercube node uses the depth first search biconnected components algorithm of [TARJ72] to partition the spanning tree edges such that two edges are in the same partition iff they are in the same biconnected component of the subgraph defined by the spanning tree edges and the edges in the adjacency matrix partition in this node.

Step 3

a) The spanning tree edge partitions in the p hypercube nodes are merged together (i.e., partitions with a common edge are combined). This is done using the standard binary processor tree. Following this the partitions are pairwise disjoint.

b) Broadcast the partitions to all hypercube nodes.

Step 4 Add the non tree edges to the remaining partitions to obtain the biconnected components.

Figure 7: Hypercube adaptation of Figure 5

hypercube run times do not include the time for the last step of Figures 2 and 7, the reported speedups and efficiencies are slightly higher than they really are. Note that difference between actual and reported figures isn't much as the last step of Figure 2 and 7 represents only a small fraction of the total time.

>From Figures 8 - 12 we conclude that the computational load of our Tarjan-Vishkin adaptation is slightly less than that of the Read modification. Also, because the Read modification has a lower communication overhead it outperforms the Tarjan-Vishkin adaptation when the ratio n/p is suitably small.

The efficiency predictions from our isoefficiency analysis are quite accurate. Going from $p = 2$ to 4, n needs to almost double to preserve efficiency; going from $p = 4$ to 8 it needs

to increase by a factor between 2 and 3; etc.

6. CONCLUSIONS

While a direct mapping of neither the Tarjan-Vishkin [TARJ75] algorithm nor the Read [READ68] algorithm is expected to perform well on a hypercube computer we are able to obtain good hypercube algorithms for the biconnected components problem by using some of ideas in these algorithms. The resulting algorithms are quite competitive and obtain a high efficiency. Our results for the biconnected components problem contrast with recently obtained results for other problems ([RANK88], [RANK89], [WOO88]) where good performance on hypercubes with a fixed number of processors could not be obtained by a suitable adaptation of the asymptotically fastest algorithms developed under the assumption that an unlimited number of processors is available.

	number of processors(p)					
size(n)	2	4	8	16	32	64
16	0.34	0.17				
32	0.51	0.30	0.15			
64	0.64	0.44	0.26	0.13		
128	0.73	0.58	0.40	0.24	0.12	
256	0.78	0.68	0.54	0.37	0.21	0.11
512	0.81	0.75	0.65	0.51	0.34	0.19
1024		0.78	0.73	0.63	0.48	0.31
2048				0.71	0.60	0.45

Figure 8: Efficiency of hypercube adaptation of the Tarjan-Vishkin algorithm (Figure 2)

7. REFERENCES

[DEKE81] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithm," SIAM Journal on Computing, 11, 4, Nov. 1981, pp. 657-675.

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.36	0.18				
32	0.51	0.31	0.17			
64	0.63	0.45	0.28	0.15		
128	0.71	0.58	0.42	0.25	0.14	
256	0.76	0.68	0.55	0.39	0.24	0.13
512	0.78	0.73	0.66	0.53	0.36	0.22
1024		0.76	0.72	0.64	0.50	0.34
2048				0.71	0.62	0.48

Figure 9: Efficiency of hypercube algorithm of Figure 7

[GOPA85] P.S. Gopalakrishnan, I.V. Ramakrishnan, and L.N. Kanal, "Computing Tree Functions on Mesh-Connected Computers," Proceedings of 1985 International Conference on Parallel Processing, 1985, pp. 703-710.

[GUST88] J. Gustafson, "Reevaluating Amdahl's Law," CACM, 31, 5, May 1988, pp. 532-533.

[HORO78] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms," Computer Science Press, Maryland, 1978.

[HORO86] E. Horowitz and S. Sahni, "Fundamentals of Data Structures in Pascal," Second Edition, Computer Science Press, Maryland, 1986.

[KUMA88] V. Kumar, V. Nageshwara, and K. Ramesh, "Parallel Depth First Search on the Ring Architecture," to appear in Proceedings of 1988 International Conference on Parallel Processing.

[LAI84] T. Lai and S. Sahni, "Anomalies in Parallel Branch and Bound Algorithms,"

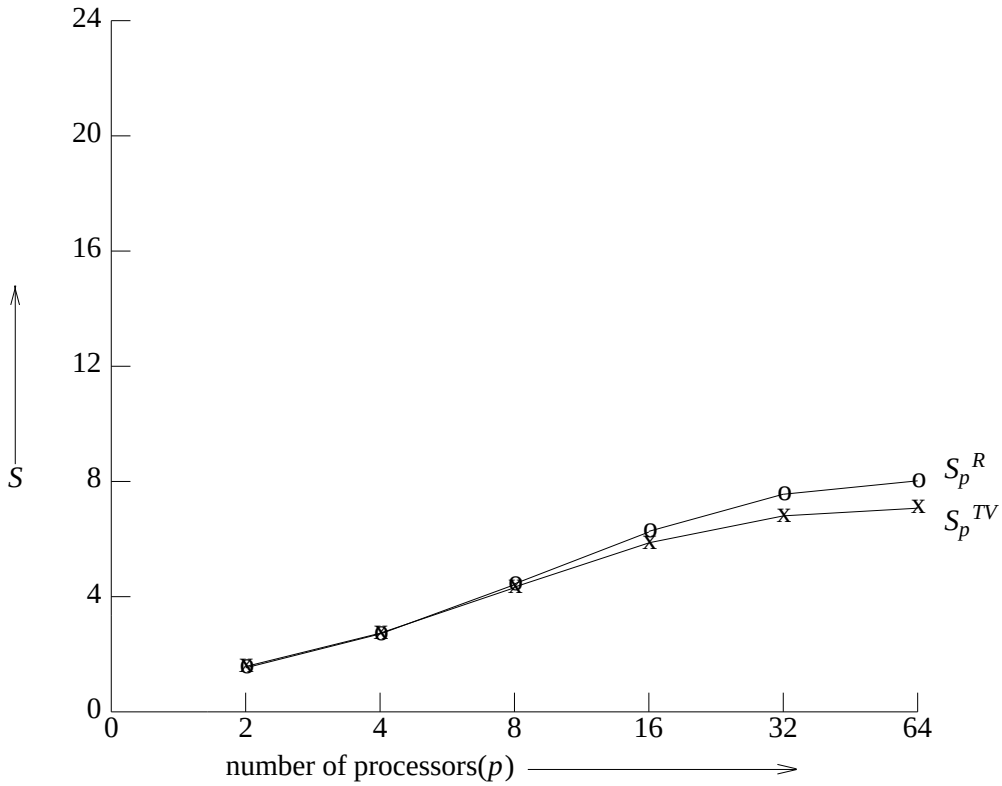


Figure 10: $n = 256$

Communications of ACM, Vol. 27, 1984, pp. 594-602.

[LI86] G. Li and B. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," IEEE Trans. on Computers, No. 6, C-35, June 1986, pp. 568-572.

[NASS80] D. Nassimi and S. Sahni, "Finding Connected Components and Connected Ones on a Mesh-connected Computer," SIAM Journal on Computing, Vol. 9, No. 4, Nov. 1980, pp. 744-757.

[NASS81] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers," IEEE Transactions on Computers, No. 2, Vol. C-30, Feb 1981, pp. 101-107

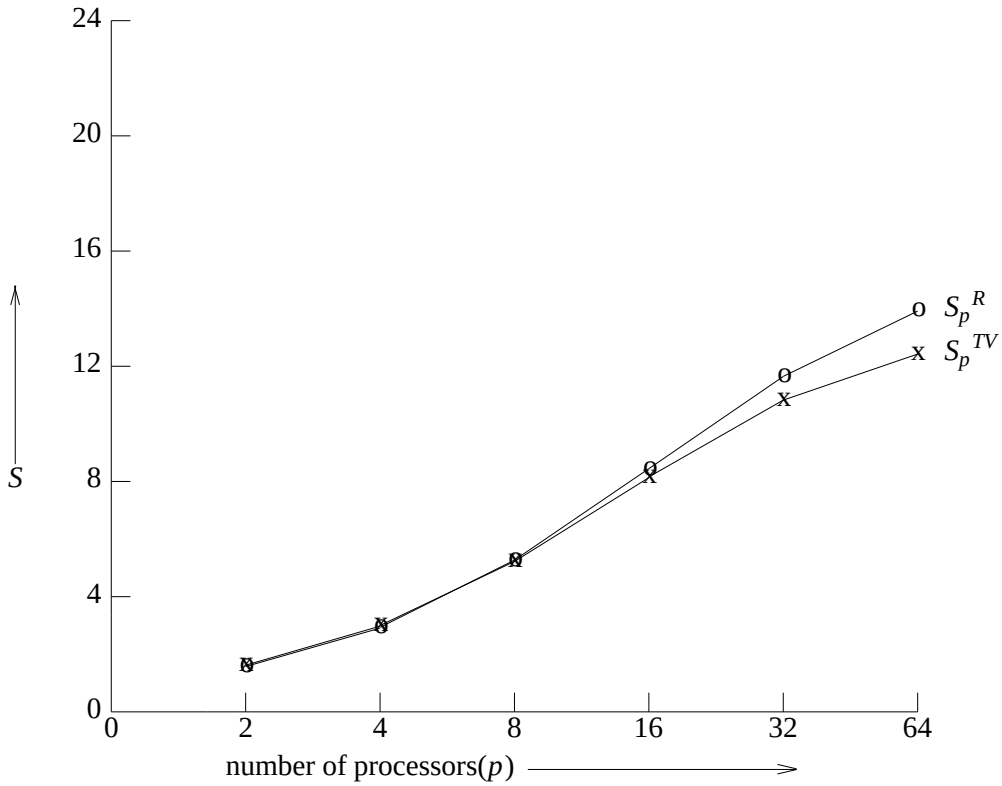


Figure 11: n = 512

[QUIN86] M. Quin and N. Deo, "An upper bound for the speedup of parallel branch-and-bound algorithms," BIT, 26, No. 1, March 1986, pp. 35-43.

[RANK88] S. Ranka and S. Sahni, "Image template matching on MIMD hypercube multicomputers," Proceedings 1988 International Conference on Parallel Processing, Vol. III, Algorithms & Applications, pp. 92-99.

[RANK89] S. Ranka and S. Sahni, "Computing Hough transforms on hypercube multicomputers," University of Minnesota, Technical Report 89-1.

[READ68] R. Read, "Teaching Graph Theory to a Computer," Waterloo Conference on

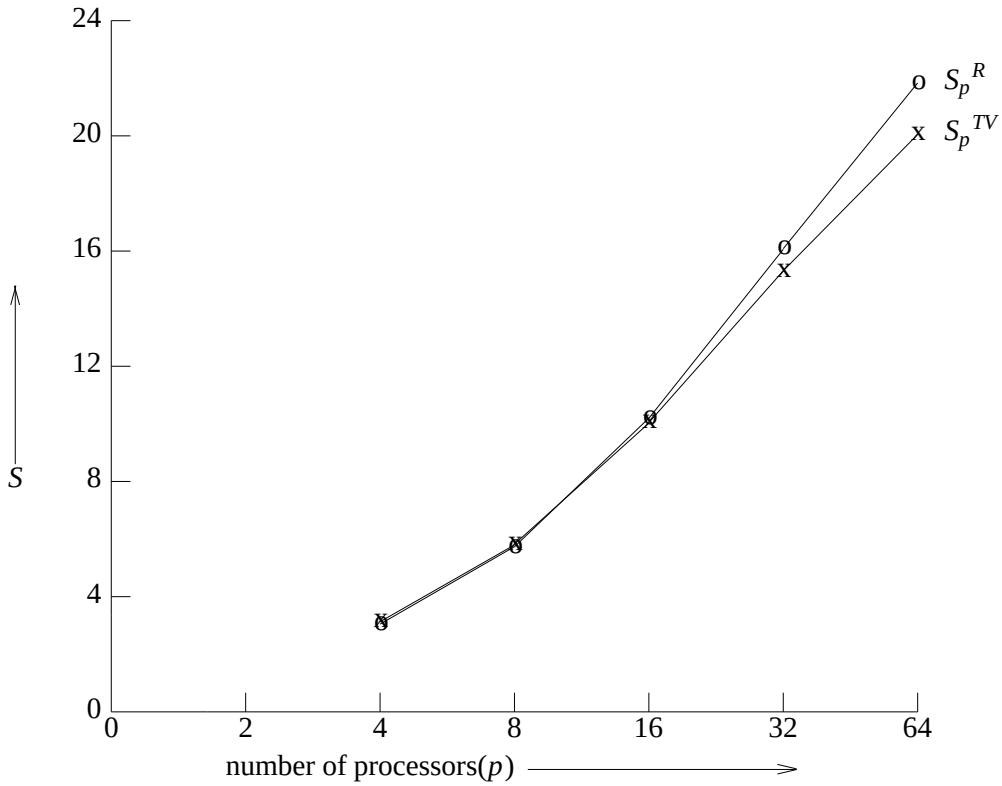


Figure 12: $n = 1024$

Combinatorics (3rd: 1968), in Recent Progress in Combinatorics, ed. by W. Tutte, Academic Press, 1969, pp.161-173.

[SHIL82] Y. Shiloach and U. Vishkin, "An O(log n) Parallel Connectivity Algorithm," Journal of Algorithms, 3, 1982, pp. 57-67.

[TARJ72] R. Tarjan, "Depth First Search and Linear Graph Algorithms," SIAM Journal on Computing, Vol. 1, 1972, pp. 146-160.

[TARJ85] R. Tarjan and U. Vishkin, "An Efficient Parallel Biconnectivity Algorithm," SIAM Journal on Computing, Vol. 14, No. 4, Nov. 1985, pp. 862-874.

[WOO88] J. Woo and S. Sahni, "Hypercube Computing: Connected Components," IEEE Proceedings of Workshop on the Future Trends of Distributed Computing Systems in the 1990s, 1988, pp. 408-417.

[WYLL79] J. Wyllie, "The Complexity of Parallel Computation," Technical Report TR79-387, Dept. Computer Science, Cornell Univ., Ithaca, NY, 1979.