HYPERCUBE COMPUTING: CONNECTED COMPONENTS*

Jinwoon Woo and Sartaj Sahni

University of Minnesota

Abstract

Several approaches to finding the connected components of a graph on a hypercube multicomputer are proposed and analyzed. The results of experiments conducted on an NCUBE hypercube are also presented. The experimental results support the analysis.

Keywords and phrases:
Hypercube computing, MIMD computer, parallel programming, connected components

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

1. INTRODUCTION

The problem of finding the connected components of an undirected graph arises in several applications. One of these is net extraction from circuit masks. A circuit mask may be modeled by an undirected graph in which the vertices represent mask polygons and edges join pairs of polygons that overlap. The connected components of the resulting graph represent the nets of the circuit realized by the mask. Once these nets have been extracted, they may be compared with a known correct set of nets to verify the correctness of the mask. The number of polygons in large masks is of the order of one million. Consequently net extraction takes a lot of time on conventional computers. In case an error is found in the mask, the mask is corrected and net extraction done again. This further increases the overall time spent verifying circuit masks. Because of this, the connected components problem is a good candidate for solution on a multicomputer.

In this paper we explore several ways to compute the connected components of a graph starting from its adjacency matrix representation. The objective is to develop an efficient algorithm for a hypercube multicomputer with a fixed number of processors. The algorithms we propose are first analyzed using conventional measures such as asymptotic complexity, speedup, and efficiency and also using a recently proposed measure isoefficiency [Kumar et al. 1988]. The proposed algorithms are then evaluated experimentally on an MIMD hypercube multicomputer. A block diagram of such a computer is given in Figure 1.1. The multicomputer has a host processor with local memory. The hypercube is attached to this host much like a peripheral device. Each hypercube processor (called node) has its own local memory. The hypercube is MIMD and all interprocessor communication and synchronization is done by explicit message passing. A program typically consists of a subprogram that runs on the host together with subprograms for each of the hypercube nodes. Often, the same subprogram is run on each node.

Our analysis of various parallel connected component algorithms shows that good performance cannot be expected by adapting the asymptotically efficient algorithms of [Dekel et al. 1981, Hirschberg et al. 1979, and Shiloach and Vishkin 1982]. Rather to obtain good performance we need to use a parallel algorithm that does a total amount of work comparable to that done by the fastest uniprocessor algorithm.

Programming a multicomputer requires one to consider several factors that do not arise when one is programming a conventional uniprocessor computer. When programming a typical conventional computer, the initial algorithmic abstraction one begins with is, perhaps, the only significant consideration. For a multicomputer, however, many other factors can have considerable impact on the efficiency of the final program. Some of these are ([Geist and Heath 1986 and Ranka et al. 1988]):

1. Algorithm Selection
2. Partitioning and Mapping
3. Overlapping Computation and Communication
4. Load Balancing

5. Using the Host

Our development of hypercube algorithms for the connected components problem is organized around these factors. Before proceeding to the development of the connected component algorithms, we describe, in section 2, the various measures used to evaluate multicomputer programs and algorithms.

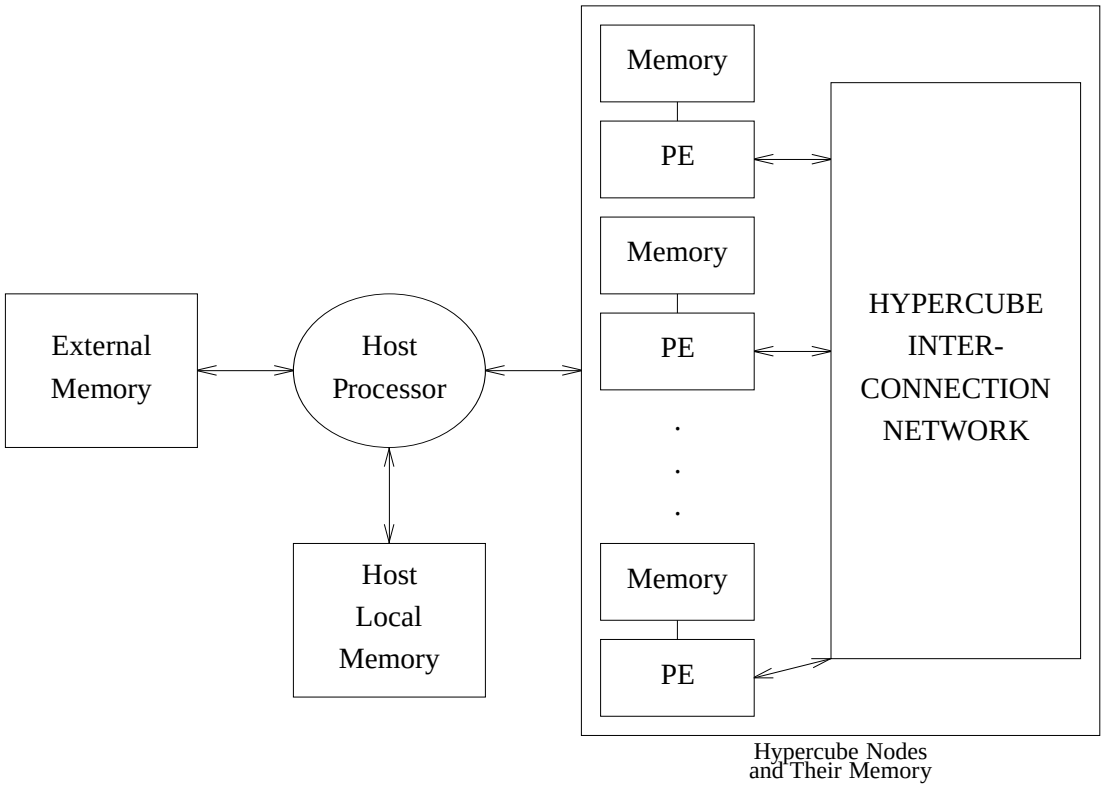


Figure 1.1

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 (as is our hypercube of Figure 1.1) 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 processor.

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 [Kumar et al. 1988, Lai and Sahni 1984, Li and Wah 1986, and Quin and Deo 1986], the speedup will be between 0 and p and the efficiency between 0 and 1. To understand the source of anomalous behavior that results in $S_p > p$ and $E_p > 1$, consider the search tree of Figure 2.1. The problem is to search for a node with the characteristics of C. The best uniprocessor algorithm (i.e., the one that works best on most instances) might explore subtree B before examining C. A two processor parallelization might explore subtrees B and C in parallel. In this case, $t_2 = 2$ (examine A and C) while $t_0 = k$ where $k-1$ is the number of nodes in subtree B. So, $S_2 = k/2$ and $E_2 = k/4$.

One may argue that in this case t_0 is really not the smallest uniprocessor time. We can do better by a breadth first search of the tree. In this case, $t_0 = 3$, $t_2 = 2$, $S_2 = 1.5$, and $E_2 = 0.75$. Unfortunately, given a search tree there is no known method to predict the optimal uniprocessor search strategy. Thus in the example of Figure 2.1, we could instead be looking for a node D that is at the bottom of the leftmost path from the root A. So, it is customary to use for t_0 the run time of the algorithm one would normally use to solve that problem on a uniprocessor.

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 [Gustafson 1988]. An interesting property of a parallel

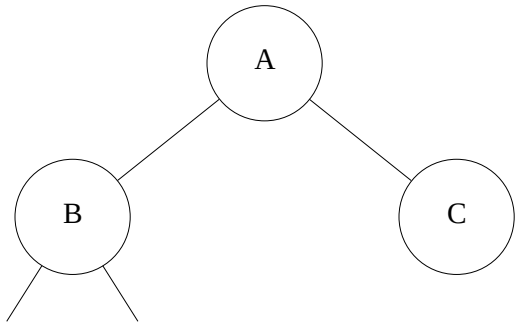


Figure 2.1

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 et al. 1988] 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 when $p \leq n^2$,

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

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.

3. ALGORITHM SELECTION

As mentioned in [Ranka et al. 1988] the algorithmic abstraction that we begin with has a significant impact on the resulting hypercube program. The starting point of the program development process could be an existing parallel algorithm developed under the assumption that an unlimited number of processors are available, a parallel algorithm developed for a fixed number of processors, or some uniprocessor algorithm that has yet to be parallelized. In the best of situations, the development of a hypercube program would begin with a parallel hypercube algorithm developed for a fixed number of processors. We know of no such algorithm for the connected components problem.

Many researchers have developed parallel connected component algorithms under the assumption that an unlimited number of processors are available. [Carlson 1987, Gopalakrishnan et al. 1985, Hirschberg et al. 1979, Huang 1985, Nassimi and Sahni 1980, and Shiloach and Vishkin 1982] are some examples of such research. None of these algorithms provides a suitable starting point for our work. For example, consider the algorithm of [Shiloach and Vishkin 1982]. Their algorithm finds the connected components of an undirected graph with n vertices and e edges in time $O(\log n)$ using a CRCW shared memory computer with $O(n+2e)$ processors. This may be run on an $O(n+2e)$ processor hypercube by using the $O(\log^2 n)$ random access read and write algorithms of [Nassimi and Sahni 1981]. The complexity of the resulting hypercube algorithm is $O(\log^3 n)$. On a uniprocessor, the connected components can be found in $O(n+e)$ time, using either depth or breadth first search. For dense graphs, $e = O(n^2)$ and the speedup, $S_{p=n^2} = O(n^2/\log^3 n)$ (In all our speedup computations we shall use $t_0 = n^2$. This is justified as we assume an adjacency matrix representation. Even if an edge

representation is used we can justify this by restricting ourselves to dense graphs). The efficiency $E_{p=n^2}$ is $O(1/\log^3 n)$. Hence, efficiency declines to zero as p (and hence n) increase.

The processor-time product is a measure of the total work (useful and nonuseful) done by a parallel algorithm. The processor-time product of the $O(n+2e)$ processor hypercube simulation of the algorithm of [Shiloach and Vishkin 1982] is $O((n+2e)\log^3 n)$. For a dense graph, this is $O(n^2\log^3 n)$. The uniprocessor algorithm does only $O(n^2)$ work. If we assume the constants of proportionality are the same in both cases, then the parallel algorithm is doing $\log^3 n$ times more work. Hence, if $n = 1024$, then it would take $\log^3 n = 1000$ processors just to break even with the uniprocessor algorithm running on a single processor. In practice, many more processors would be needed to break even as the constant of proportionality is much larger for the Shiloach-Vishkin hypercube adaptation (this comes from the increased constant factor for their algorithm; the constant factor associated with random access reads and writes; and the need for interprocessor communication which is typically far more expensive per unit than a basic arithmetic).

Dekel, Nassimi, and Sahni [Dekel et al. 1981] have developed an $O(\log^2 n)$ hypercube algorithm to find a spanning forest of an n vertex graph. This uses $n^3/\log n$ processors. This algorithm may be adapted to find connected components in $O(\log^2 n)$ time. The processor-time product of this adaptation is $O(n^3\log n)$. For $n = 1024$, approximately $n\log n = 10240$ processors are needed to break even with the uniprocessor algorithm running on a single processor computer.

The starting point for our hypercube program is the relatively simple low overhead algorithm given in Figure 3.1. This assumes a dense graph and an adjacency matrix representation. Each hypercube processor begins with a partition of the adjacency matrix. It computes a spanning forest under the assumption the graph has only those edges that are in its partition. The first step of this algorithm is the same as the data reduction step in the connected component algorithm proposed by Huang [Huang 1985] for the mesh-of-trees multicomputer. The details of the algorithm for step 1 are provided in Figure 3.2 (procedure *Spanning Forest*). The input to this procedure consists of the vertices V_r represented by the rows of the adjacency matrix partition in the hypercube node and the vertices V_c represented by the columns of this partition. The procedure uses a breadth first traversal [Horowitz and Sahni 1986]. A depth first traversal could also have been used.

The spanning forests define a relationship, **R**, between pairs of vertices. $i\mathbf{R}j$ iff i and j are in the same tree in at least one forest. The transitive closure of this relation may be computed using the union-find scheme discussed in [Horowitz and Sahni 1986]. This partitions the vertices into equivalence classes. Each such class defines a connected component. We shall refer to the process that results in the transitive closure of **R** as *spanning structure merging*. Define a *spanning structure* to be a collection of trees with the property that every graph vertex is in exactly one tree and if two vertices are in the same tree then they are in the same connected component of the graph. Note that the edges in a spanning structure are not required to be graph edges. Each of the p spanning forests computed in step 1 of Figure 3.1 is

Step 1: Each hypercube processor computes a spanning forest based on the information in its adjacency matrix partition. This is done using breadth first or depth first search.

Step 2: The hypercube processors merge their spanning forests to obtain the connected components.

Figure 3.1: The connected components algorithm

a spanning structure. In step 2 we begin with these p spanning structures and combine them pairwise (say) until just one spanning structure remains. This final spanning structure has the property that two vertices are in the same tree iff they are in the same connected component. An example illustrating the two steps in our algorithm is given in Figure 3.3. For this example the final spanning structure will consist of two trees; one with vertices 1, 2, 3, and 4 and the other with vertices 5 and 6. Figure 3.3(e) shows just one of the possible spanning structures with this property. The correctness of the algorithm of Figure 3.1 follows from the observation that in step 1 only edges that are on cycles are eliminated. This does not affect the connected components.

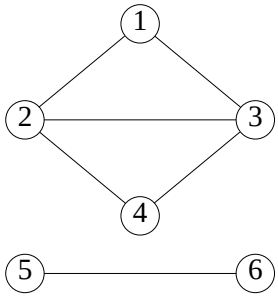
Step 1 requires a total $O(n^2)$ work. Since a spanning structure is a collection of trees, it can have at most $n-1$ edges. Combining two such structures takes slightly more than linear time if the union-find scheme of [Horowitz and Sahni 1986] is used. It takes linear time if the equivalence class algorithm of [Horowitz and Sahni 1986] is used. However, the latter scheme requires more memory. This becomes a problem in our case when testing with large graphs. So we do not use it. Since $p-1$ pairwise merges of spanning structures are performed in step 2 the total work done in this step is $O(np\alpha(n))$ where $\alpha(n)$ accounts for the fact that union-find takes slightly more than linear time (α is a functional inverse of the Ackermann's function). Since p is assumed fixed (or alternatively if we assume $p\alpha(n) \leq n$), the total workload of Figure 3.1 is $O(n^2)$. So this has a better potential of exhibiting good speedup for "small" p than the algorithms of [Shiloach and Vishkin 1982] and [Dekel et al. 1981].

```
Procedure Spanning Forest (V_r, V_c) ;
{Find spanning forest edges for the partition with row vertices V_r and column vertices V_c}
    initialize queue empty;
    initialize all n vertices to be unmarked;
    for each vertex l in V_r do
        if vertex l is unmarked then {find a tree for l}
        begin
            add l to queue and mark it;
            while queue not empty do
            begin
                delete first vertex (say j) from queue;
                if j ∈ V_r
                then scan row for vertex j
                else if j ∈ V_c then scan column for vertex j;
                all unmarked vertices k encountered during this scan are marked,
                edge (j,k) is output as part of the spanning forest, vertex k is added to
                the queue.
            end; {of while}
        end; {of then and for}
    end; {Spanning Forest}
```

Figure 3.2: Finding the spanning forest for an adjacency matrix partition

4. PARTITIONING AND MAPPING

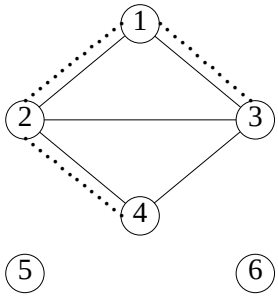
In this section, we shall consider two refinements of the algorithm of Figure 3.1. In both, steps 1 and 2 are done in sequence (i.e., step 2 commences after step 1 has completed). In a later section, we consider another refinement in which steps 1 and 2 are done in parallel. The two partitioning schemes of this section were used in [Jenq and Sahni 1987] for the all pairs shortest paths problem. Since in our hypercube model the memory is distributed across the nodes of the hypercube and it takes less time for a node to access its local memory than that of another node, it is necessary to distribute the adjacency matrix across the processor memories. The distribution schemes studied here, in effect, partition the matrix. However, a partitioning isn't always as effective as a data distribution scheme that allows some data replication [Ranka et al. 1988]. Along with a data partitioning, one needs to provide a mapping of



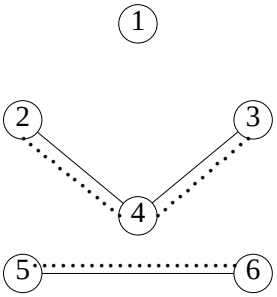
(a) graph G

1	0	1	1	0	0	0	
2	1	0	1	1	0	0	G_1
3	1	1	0	1	0	0	
4	0	1	1	0	0	0	
5	0	0	0	0	0	1	G_2
6	0	0	0	0	1	0	

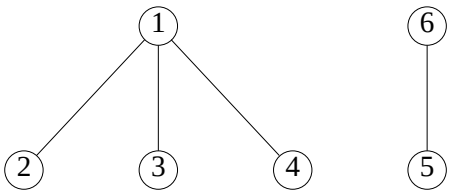
(b) adjacency matrix



(c) subgraph G_1 and spanning forest F_1



(d) subgraph G_2 and spanning forest F_2



(e) possible spanning structure after merging F_1 and F_2

Figure 3.3

the data partitions to the processor memories.

4.1 Partitioning By Stripes

In this case, an $n \times n$ adjacency matrix is partitioned into p stripes with each stripe comprised of n/p contiguous rows. Figure 4.1 shows the partitioning and processor mapping for the case $n = 32$ and $p = 8$. In this figure, P_i denotes processor i of the hypercube.

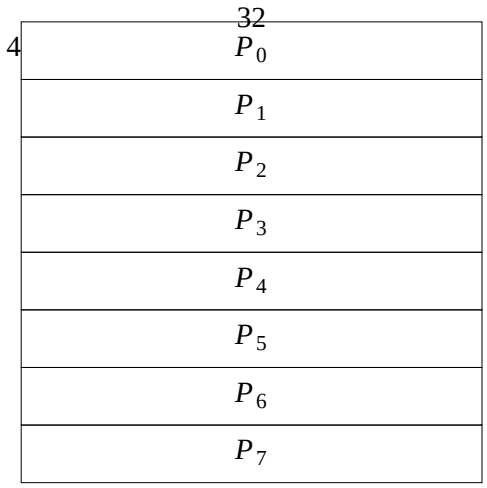


Figure 4.1: Partitioning into stripes and processor mapping

To compute the connected components, each processor first computes a spanning forest of the given n vertex graph. This spanning forest is computed using procedure *Spanning Forest* (Figure 3.2) with $V_r = \{(i-1)n/p, \cdots, in/p-1\}$ for processor P_i and $V_c = \{0,1, \cdots, n-1\}$ for all processors. The merging of the spanning structures is done pairwise as indicated in Figure 4.2 for the case $p = 8$. Figure 4.3 shows the hypercube communication paths. Processor P_0 is involved in three stages of merging. First, it merges its step 1 structure with that of P_1. For this, P_1 must transmit its spanning structure information to P_0. Next, it merges this spanning structure with the merge of the step 1 spanning structures of P_2 and P_3. For this, P_2 communicates appropriate information to P_0. Finally, P_0 merges the merged step 1 spanning structure of P_0 through P_3 with that of P_4 through P_7. The overall spanning structure resides in P_0. At this point, each vertex determines the root of the spanning structure tree it is contained in. This is its connected component identifier. Notice that

the number of active processors reduces by half following each merge step.

When merging two spanning structures A and B we take the at most $n-1$ edges in one (say A) and merge with those of the other (say B). For each of the edges in A two finds are performed to see if the two vertices that are the end points of this edge are already in the same tree of B. If they are not then the two trees of B that contain these vertices are unioned. If the two vertices are in the same tree of B no union is performed. Hence a pairwise merge requires at most $2(n-1)$ finds and $n-1$ unions.

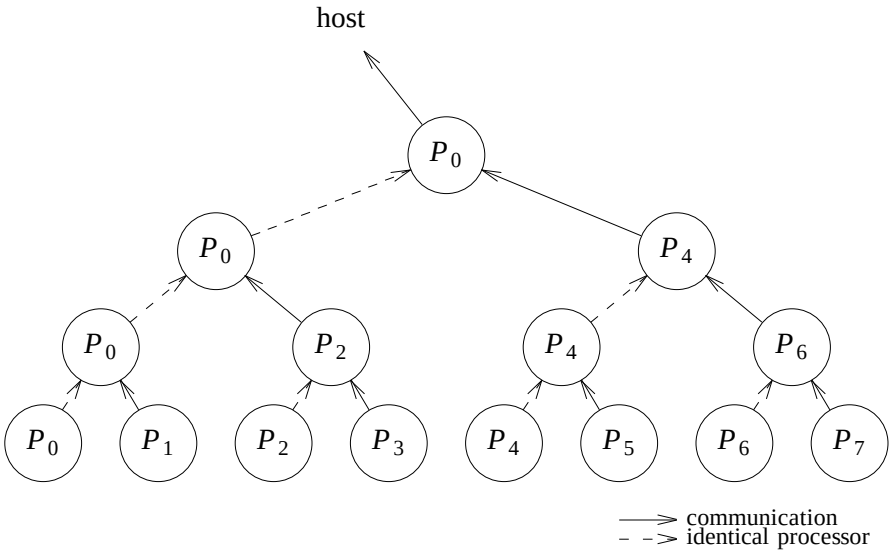


Figure 4.2: Communication path

Since a processor's adjacency matrix partition has n/p rows of n bits each, step 1 takes $O(n^2/p)$. More accurately, in the worst case the n/p rows of the partition will be scanned in the **then** clause of Figure 3.2 and the $n-n/p$columns that correspond to the $n-n/p$ vertices in V_c-V_r scanned in the **else** clause. So, a total of $n^2/p+(n-n/p)n/p = 2n^2/p-n^2/p^2$ accesses to the processor's adjacency matrix partition are made. Hence, step 1 takes $n^2/p(2-1/p)t_s$ time (t_s is a constant). There are log$p$ merge stages with each taking at most $(n-1)t_m$ time (for simplicity, we assume that $2(n-1)$ finds and $n-1$ unions can be done in $O(n)$ time; the union-find algorithms described in [Horowitz and Sahni 1986] take slightly more time; linear time can be achieved using the equivalence class algorithm of [Horowitz and Sahni 1986]; t_m is a

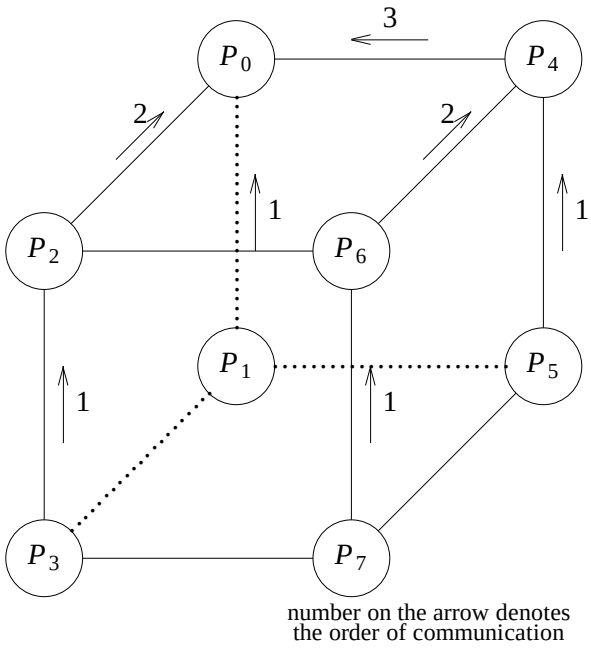


Figure 4.3: Communication path on hypercube

constant). Note that a spanning structure of an n vertex graph/subgraph can contain at most $n-1$ edges. Each communication of a spanning structure takes at most $\alpha + (n-1)t_c$ worst case time where α is the communication startup time and t_c is a constant. The overall worst case time (excluding the final component identification time) is :

$$t_{stripes} = n^2/p(2-1/p)t_s + (n-1)t_m \log p + (n-1)t_c \log p + \alpha \log p$$

Since t_0 is $n^2 t_s$, the speedup $S_p^{stripes}$ is

$$S_p^{stripes} = \frac{n^2 t_s}{t_{stripes}}$$

$$= \frac{n^2 t_s}{n^2/p(2-1/p)t_s + (n-1)t_m \log p + (n-1)t_c \log p + \alpha \log p}$$

and the efficiency $E_p^{stripes}$ is

$$E_p^{stripes} = \frac{1}{2 - \frac{1}{p} + \frac{p}{n^2 t_s}\left[(n-1)(t_m + t_c) + \alpha\right] \log p}$$

For constant efficiency, we require

$$\frac{p}{n^2 t_s}\left[(n-1)(t_m+t_c)+\alpha\right]\log p - \frac{1}{p}$$

to be constant. For this, the problem size, n, must grow at rate $\Omega(p\log p)$. The work load, n^2, must therefore grow at the rate $\Omega(p^2\log^2 p)$. Hence, the isoefficiency is $\Omega(p^2\log^2 p)$. >From the equation for $S_p^{stripes}$, we get

$$S_p^{stripes} < \frac{n^2 t_s}{\frac{n^2}{p}(2-1/p)t_s} = \frac{p^2}{2p-1}$$

Hence, $E_p^{stripes} < \frac{p}{2p-1}$, for graphs which require the examination of all n/p rows of V_r and $n-n/p$ columns of $V_c - V_r$ in step 1. These graphs have the property that in at least one stripe each vertex in $V_c - V_r$ is adjacent to at least one vertex in V_r. Note that as the edge density increases, the probability of this happening also increases. Further if this property is satisfied, the graph is connected. However, a connected graph need not satisfy this property. For graphs with this property, $E_p^{stripes} < 2/3$ for $p = 2$, $4/7$ for $p = 4$, $8/15$ for $p = 8$, etc. Note that on graphs that do not satisfy the stated property the efficiency can be higher. For sufficiently dense graphs these bounds can be expected to apply as such graphs satisfy the above property.

4.2 Partitioning By Rectangles

Partitioning by rectangles is an alternate to partitioning by stripes. The adjacency matrix is partitioned into p rectangles of size $\frac{n}{2^{\lceil d/2\rceil}}\times\frac{n}{2^{\lfloor d/2\rfloor}}$ where $p = 2^d$. Figure 4.4 shows the partitioning and processor mapping for the case $n = 32$ and $p = 8$. The mapping is designed to optimize the spanning structure mergings of step 2. While this partitioning is the same as that used in [Jenq and Sahni 1987] for the all pairs shortest paths problem, the mapping to processors is different. The spanning structure merge order is shown in Figure 4.5. This merge order minimizes the spanning structure size following each merge.

The worst case step 1 time for this scheme is $2n^2/p\, t_s$ as in the worst case $|V_c - V_r| = |V_c| = \frac{n}{2^{\lfloor d/2\rfloor}}$ and all $\frac{n}{2^{\lceil d/2\rceil}}$ rows are scanned in the **then** clause and all $\frac{n}{2^{\lfloor d/2\rfloor}}$ columns in the **else** clause. The communication and merge time is a function of the number of stages and the number of edges being merged or communicated. For simplicity, we assume that the number of edges in the spanning structure corresponding to an $i\times j$ partition is at most $2i$ ($i\geq j$) even though $i+j-1$ is a better bound. When d is even, at most $x = 2*\frac{n}{2^{d/2}}$ edges are transmitted by each of the processors that transmit edges; at most $2x$ edges are transmitted in each of the next two stages; at most $4x$ in the next two stages; ...; and at most $2^{d/2}x$ in the last stage. So, when d is even, at most

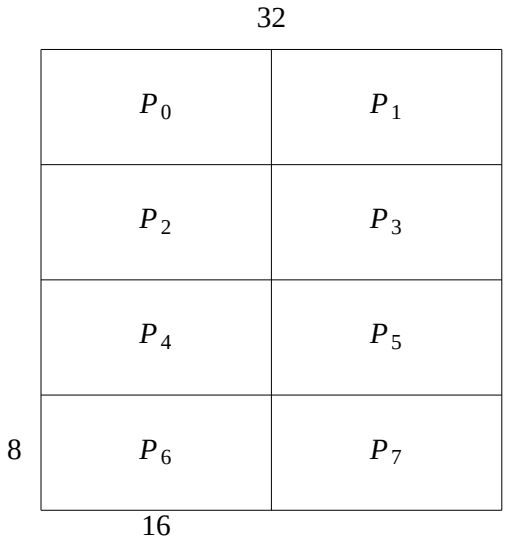


Figure 4.4: Partitioning into rectangles and processor mapping

$$x(1+2+2+4+4+\cdots+2^{d/2}) = 2\frac{n}{2^{d/2}}(1+4+8+\cdots+2^{d/2})$$

$$= \frac{2n}{2^{d/2}}(1+2+4+8+\cdots+2^{d/2}-2)$$

$$< 6n$$

edges are transmitted. Note that $2^{d/2}x = 2n > n-1$. However a spanning structure can have at most $n-1$ edges. So, the above bound is quite loose.

A similar analysis shows that $6n$ bounds the total data transmission when d is odd. Also, since at most $n-1$ edges may be in a spanning structure, we get $\min\{6n,(n-1)\log p\}$ as a bound on the total number of edges transmitted by any one node. Hence the worst case time complexity is:

$$t_{rectangles} = \frac{2n^2}{p}t_s + \min\{6n,(n-1)\log p\}t_m + \min\{6n,(n-1)\log p\}t_c + \alpha\log p$$

Comparing with $t_{stripes}$, we see that the worst case step 1 time for the stripes method is less than that for the rectangles method by n^2/p^2. The step 2 time for the stripes method is

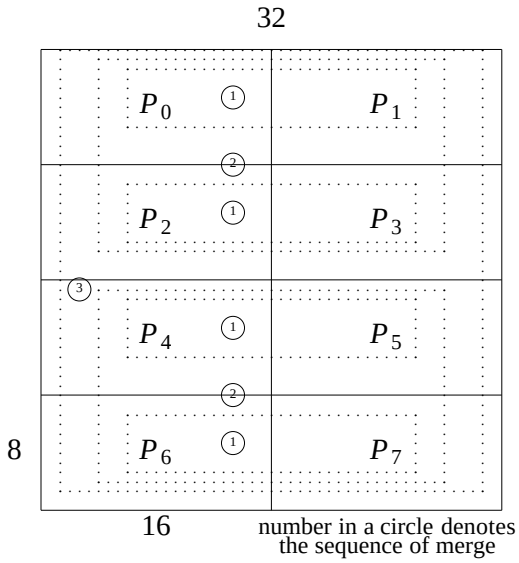


Figure 4.5: Merging sequence of rectangles

never less than that of the rectangles method. In fact, when $6n < (n-1)\log p$ (or approximately when $p > 64$), the step 2 time for the rectangles method is less than that for the stripes method. As noted earlier, our $6n$ bound is quite loose and we expect the step 2 time for the rectangles method to be less than that for the stripes method even when p is less than 64.

The speedup and efficiency for the rectangles method are:

$$S_p^{rectangles} = \frac{n^2 t_s}{t_{rectangles}}$$

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

$$= \frac{1}{2 + \frac{p}{n^2 t_s}\left[\min\{6n, (n-1)\log p\}(t_m + t_c) + \alpha \log p\right]}$$

For constant efficiency,

$$\frac{p}{n^2 t_s}\left[\min\{6n, (n-1)\log p\}(t_m + t_c) + \alpha \log p\right]$$

must be constant. When $\min\{6n,(n-1)\log p\} = (n-1)\log p$, the isoefficiency is the same, $\Omega(p^2\log^2 p)$, as that of the stripes method. When $\min\{6n,(n-1)\log p\} = 6n$,

$$\frac{p}{n^2 t_s}\left[6n(t_m+t_c)+\alpha\log p\right]$$

must be constant. For $n \gg \log p$, this requires that n grows as $\Omega(p)$. Hence the work load, n^2, must grow as $\Omega(p^2)$. Hence the isoefficiency of the rectangles method is between $\Omega(p^2)$ and $\Omega(p^2\log^2 p)$.

>From the equation for $E_p^{rectangles}$ we see that for worst case data, $E_p^{rectangles} < 1/2$. We expect this bound to apply for sufficiently dense graphs.

4.3 Experimental Results

FORTRAN programs to find connected components using the stripes and rectangles partitioning schemes were run on an NCUBE hypercube multicomputer. For each n, 30 random graphs with edge density ranging from 70% to 90% were generated. The average efficiency is given in the tables of Figures 4.6 (stripes partitioning) and 4.7 (rectangle partitioning). As predicted by our isoefficiency analysis, the problem size n needs to more than double each time the number of processors doubles in order for the efficiency to not deteriorate. For example, the stripes method has an efficiency 0.2 when $n = 64$ and $p = 8$. To get this same efficiency when $p = 16$, we need n to be greater than 128. The problem size increase required by the rectangles method is not as great as required by the stripes method (though still n must more than double each time p doubles).

Our analysis indicated that the efficiency would be less than 2/3 for the stripes method when the test graphs required the examination of all rows of V_r and all columns of $V_c - V_r$. The table of Figure 4.6 has a few entries with efficiency greater than this. This indicates that our average test graph did not require all these rows and columns to be examined. While not shown in the table, we observed that the efficiency became closer to that predicted by our analysis as the edge density was increased. As p increases, the efficiency declines because of an increase in the inter processor communication overhead. For the rectangles method, again, some efficiences exceed the 0.5 bound expected for worst case data. This reflects the fact that our test graphs were not worst case graphs.

Also, our analysis indicated that the step 1 time for the stripes method is less than that for the rectangles method. However, the step 2 time for the rectangles method is generally less. This differential in step 2 time increases with p. Hence, we expect the stripes method to outperform the rectangles method for large n and small p. This expectation is reflected in the data of Figures 4.6 and 4.7. The speedup obtained by the two methods for n = 256, 512, and 1024 are plotted in Figures 4.8 through 4.10.

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.45	0.21				
32	0.53	0.28	0.13			
64	0.61	0.36	0.20	0.10		
128	0.67	0.46	0.29	0.15	0.08	
256	0.71	0.54	0.38	0.24	0.13	0.06
512	0.73	0.59	0.47	0.34	0.21	0.11
1024		0.62	0.53	0.43	0.30	0.18
2048				0.49	0.39	0.27
4096						0.35

Figure 4.6: Efficiency of stripes partitioning

5. OVERLAPPING COMPUTATION AND COMMUNICATION

The refinements of the preceding section make no attempt to overlap the time spent computing with that spent transmitting data. Figure 5.1 shows the activities of processor p_0 of Figure 4.2. We can attempt to reduce the overall time by overlapping data transmission and computation. For this, the odd numbered leaf processors of Figure 4.2 must transmit their spanning structure edges in packets concurrent with the computation of the spanning structure. If we are sending packets of size s edges, $s < n$, then as soon as s structure edges are selected, a transmit is initiated. This requires a slight modification in the merging process so that it commences as soon as the first packet is received. Similarly, during each merging step, the merged structure is transmitted as a series of packets.

If $n-1$ edges are to be transmitted in packets of size s, then the total transmission time becomes $(n-1)(\alpha+st_c)/s$. While this is larger than the $\alpha+(n-1)t_c$ time needed to send the $(n-1)$ edges as one packet, we can accomplish a reduction in the overall run time as the transmission may be substantially overlapped with the step 1 time and the merge times. A reduction will be seen only if the total wait time decreases.

For the connected components problem, the total wait time is $O(n\log p)$ while the computation time is $O(n^2/p)$. So, even if the wait time was reduced to zero, there would not be much difference in the overall time. Figure 5.2 shows the % change in run times of the two

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.45	0.25				
32	0.50	0.30	0.14			
64	0.57	0.36	0.20	0.13		
128	0.62	0.43	0.29	0.19	0.11	
256	0.65	0.48	0.37	0.27	0.17	0.10
512	0.67	0.51	0.44	0.36	0.25	0.17
1024		0.53	0.48	0.43	0.33	0.24
2048				0.47	0.40	0.33
4096						0.40

Figure 4.7: Efficiency of rectangles partitioning

schemes of Section 4 when the computation/communication strategy is implemented. The packet size used was 500 edges. As is evident, the overlapping strategy does not have much impact on the total run time. In fact a reduction is seen only for large n. For the stripes method we were unable to make n sufficiently large to observe a run time reduction except for the case $p = 2$. When n is large, the step 1 time is large and transmitting by packets effectively overlaps the computation of the spanning structure. When n is small, the step 1 time is small and hence not enough to reduce the wait time. It should be emphasised that in problems where the communication and computation times are comparable, successful overlapping of these can significantly reduce the overall run time. In fact, Won and Sahni [Won and Sahni 1987] report a 23% reduction in the case of the maze routing problem.

6. LOAD BALANCING

The strategy to overlap computation and communication may be taken one step further to perform steps 1 and 2 of Figure 3.1 in parallel. For this, some of the processors are assigned the task of finding spanning structures and the others the task of merging spanning structures. Since the strategy of Section 5 transmits spanning structures in packets as these

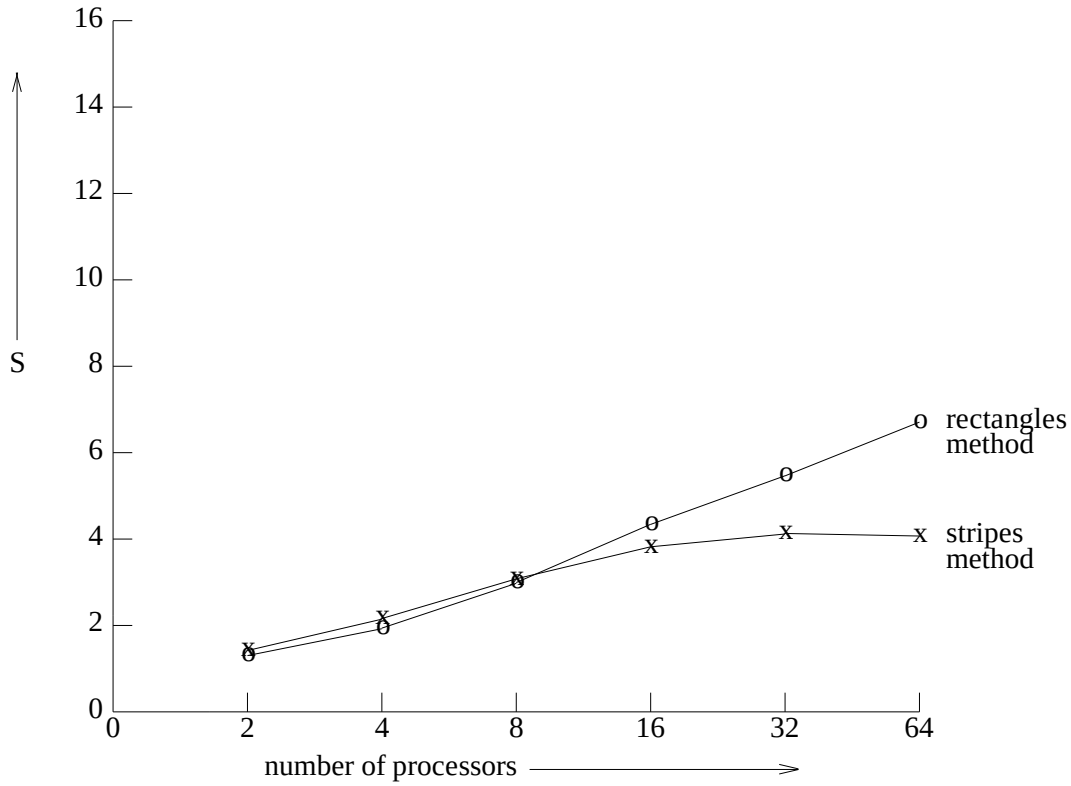


Figure 4.8: $n = 256$

are generated, it is possible for the merge processors to begin their work before the step 1 computation of a spanning structure has been completed.

Let us take a closer look at how this may be done for the stripes partition. One possibility is to partition the adjacency matrix into $\frac{n}{p} \times \frac{n}{p}$ squares as in Figure 6.1. Since the adjacency matrix of an undirected graph is symmetric, only those squares on or above the main diagonal are needed. The processors are grouped into pairs and each processor is assigned a row of squares as in Figure 6.1. The processors begin by computing a spanning structure for their diagonal square. Then the even processors transmit their structures to the odd processor in their respective pairs. The odd processors merge spanning structures while the even ones continue to process their squares (only diagonal squares and those to their right are processed) and transmit the resulting spanning structure edges to their odd partners.

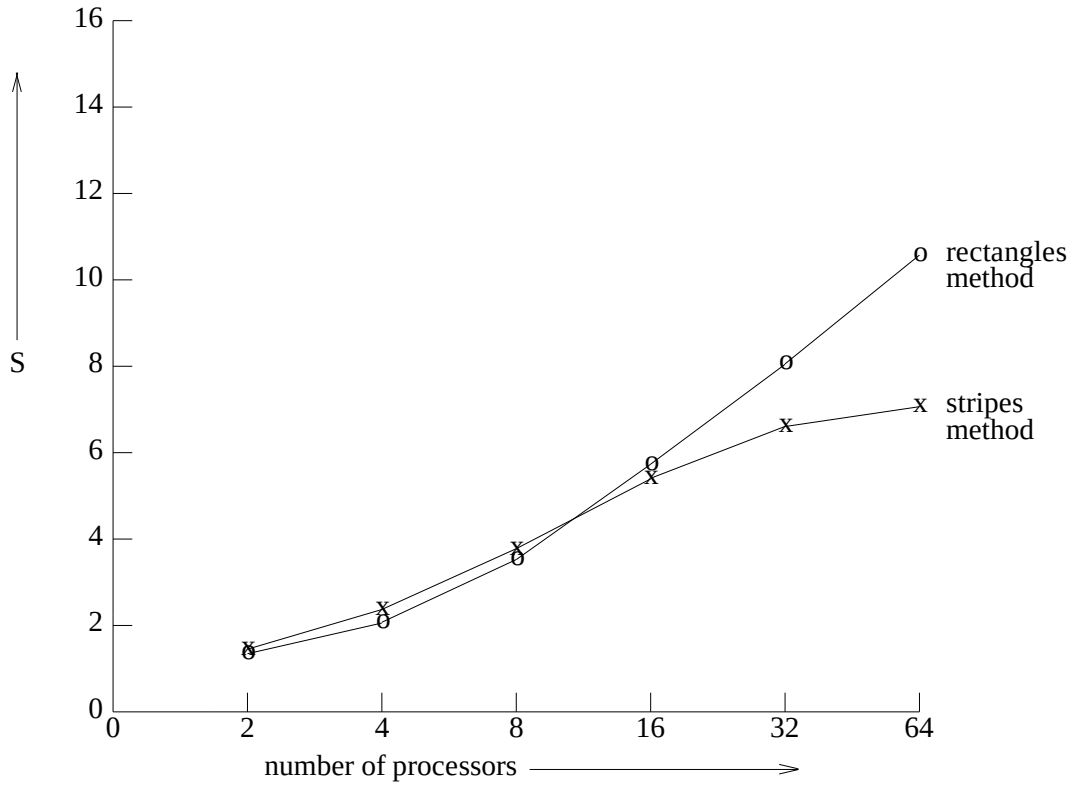


Figure 4.9: $n = 512$

This processing of squares consists of the two steps:

a) perform a breadth first traversal of the square retaining edges that form a spanning structure for the square

b) process these spanning structure edges using the union-find algorithm of [Horowitz and Sahni 1986] to eliminate edges that form a cycle when considered in conjunction with those edges already transmitted to the odd partners.

Only edges that survive step b) above are actually transmitted to the odd partners. The even processors do not transmit the spanning structures of their last q squares to their odd partners (the optimal value of q is to be determined experimentally). On completing all merges, the odd processors begin to process their squares and transmit spanning structures to their even partners. The even processors begin to merge after completing the processing of

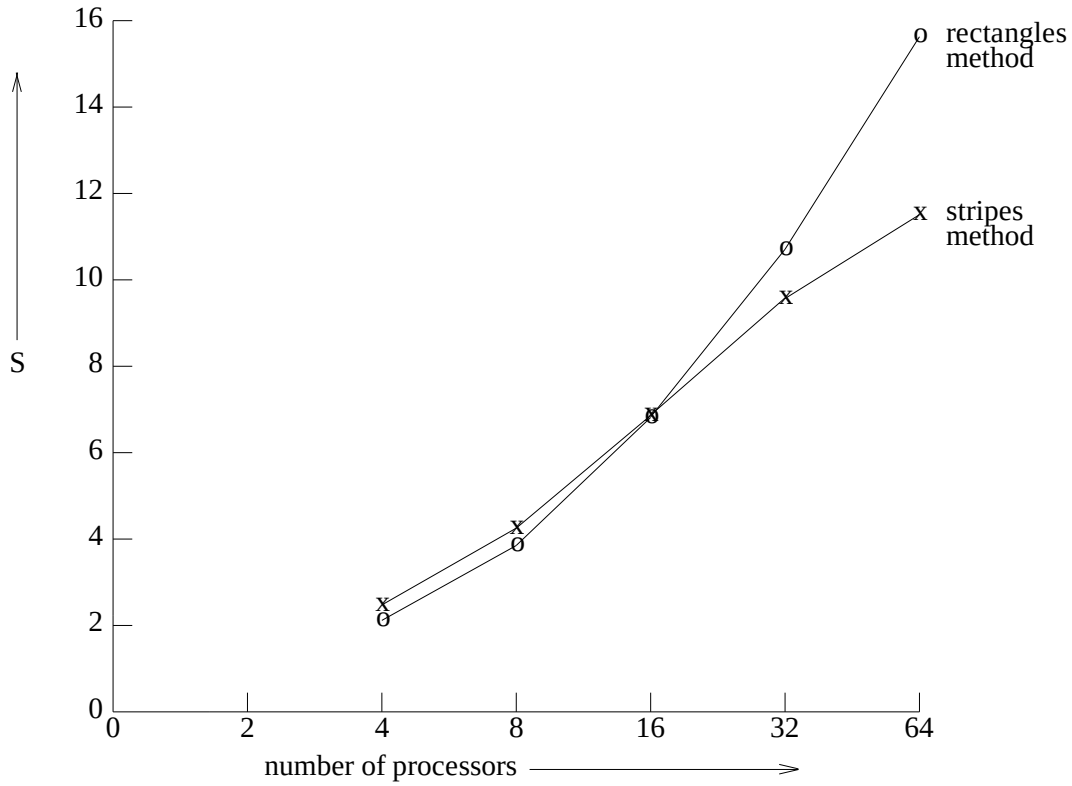


Figure 4.10: $n = 1024$

their remaining squares. Once an even processor has finished all its work, the resultant spanning structure is transmitted to the even processor in the upper adjacent pair (i.e., processor P_{2i} transmits to processor $P_{2(i-1)}$) for merging. Figure 6.2 illustrates the data transmission sequence for the case $p = 8$. Notice that since P_6 will finish first, some or all of its transmission to P_4 will be overlapped with work still being done by P_4. This overlapping could also take place between P_2 and P_4 and P_0 and P_2.

The time, $t_{squares}$, required by this strategy is given by

$$t_{squares} = T_0 + T_{wait} + T_{merge}$$

where T_0 is the time taken by P_0 to finish its work for the pair (P_0 , P_1); T_{wait} is the time P_0 has to wait following T_0 for the merged data to arrive from P_2; and T_{merge} is the time to

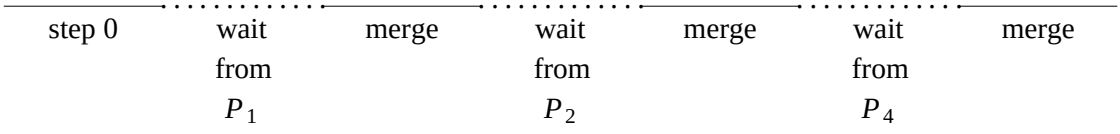


Figure 5.1: Activity status of P_0

merge this data.

The success of this strategy depends considerably on the matching of interprocessor communication times with intra processor computation times. Unfortunately, for large n, the time needed to compute the spanning structure of a square is much greater than the time needed to transmit and merge the structure. Hence the merging processors are often idle. To remedy this load imbalance, the group size may be increased to k, $k > 2$, processors. In each group of k processors, one processor merges while the remaining $k-1$ compute spanning structures. This also reduces the rightmost path length of Figure 6.2. Notice that this length is p/k. When $k > 2$, the merge processor of a group only merges structures and the adjacency matrix data for the group is distributed evenly over the remaining $k-1$ processors in the group (again, only data in the upper triangle is needed). Figure 6.3 gives the ratio $t_{squares}/t_{stripes}$ for the graphs used in the experiment of section 4.3. The number in parenthesis is the optimal value of k. Note that when $k = 2$, the pairing strategy described in the beginning of this section is used. It was experimentally determined that the best value of q is p, the number of hypercube processors. In this case the even processor in each pair obtains a spanning forest for all its squares together. The fact that $q = p$ gave the best performance may be attributed to the relative high cost of interprocessor communication. The odd processors work one square at a time and transmit edges to their even partners.

For any fixed hypercube size the optimal k increases as n increases. This is because the time needed to compute the initial spanning structures increases quadratically in n while the merge time increases linearly in n. So a merge processor can handle more merge load in the time required by the spanning structure processors to compute these structures. Because of the unpredictable nature of the computation/communication overlap in this scheme, the scheme is hard to analyze with a view to predicting the optimal k. The efficiency table is given in Figure 6.4. Since the spanning structure time increases asymptotically faster than the merge time, for

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	1.49	1.32				
32	1.15	1.18	1.09			
64	0.76	0.92	0.98	1.18		
128	0.46	0.62	0.78	0.84	0.82	
256	0.25	0.38	0.51	0.68	0.73	0.81
512	-0.17	0.21	0.35	0.46	0.60	0.72
1024		0.07	0.16	0.32	0.43	0.61
2048				0.19	0.30	0.38
4096						0.30

(a) stripes partitioning

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	1.45	1.64				
32	0.55	0.64	0.61			
64	-0.18	0.23	0.26	0.31		
128	-0.63	-0.37	0.10	0.13	0.15	
256	-1.12	-1.15	-0.53	0.00	0.12	0.15
512	-1.37	-1.43	-1.59	-0.44	-1.49	0.15
1024		-1.42	-1.40	-0.67	-0.42	-0.14
2048				-0.95	-0.68	-0.46
4096						-0.71

(b) rectangles partitioning

positive number means increase in run time.

negative number means decrease in run time.

Figure 5.2 % change in run time using packets

any fixed number p of processors the ratio $t_{squares}/t_{stripes}$ first decreases as n increases and then increases as n increases. When the ratio is decreasing the overlapping of computation and communication is the dominating factor. However, eventually the increased computation load of the squares method dominates and the ratio begins to increase.

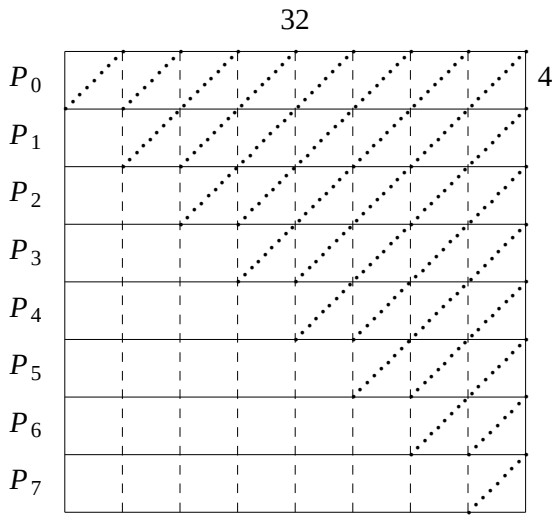


Figure 6.1: Partitioning into squares and processor mapping

The squares scheme just described uses only the upper triangles of the adjacency matrix. One may consider developing a program that does this without performing steps 1 and 2 of Figure 3.1 in parallel. In some sense, this represents the case $k = 1$ with the final merging stage replaced by a binary tree merge as in Figure 4.2. Since the number of bits in the upper triangle is $\sum_{i=1}^{n-1} i = n(n-1)/2$ (note that all diagonal bits are 0 and need not be considered), for good load balancing $n(n-1)/2p$ bits are resident with each processor initially. This also equalizes the processor memory requirements.

The worst case merging and communication time requirements of this balanced triangle scheme are the same as those of the stripes method. The worst case step 1 (cf. Figure 3.1) time is $n(n-1)/p$ as each bit in a node's partition may be examined twice. This results from a need to implement Figure 3.2 so that when a partition row is scanned, the row segment that is to the left of the diagonal is obtained by scanning the corresponding column segment above the diagonal. So, we obtain

$$t_{balanced} = n(n-1)/p\ t_s + (n-1)t_m \log p + (n-1)t_c \log p + \alpha \log p$$

>From this, the following speedup and efficiency are obtained:

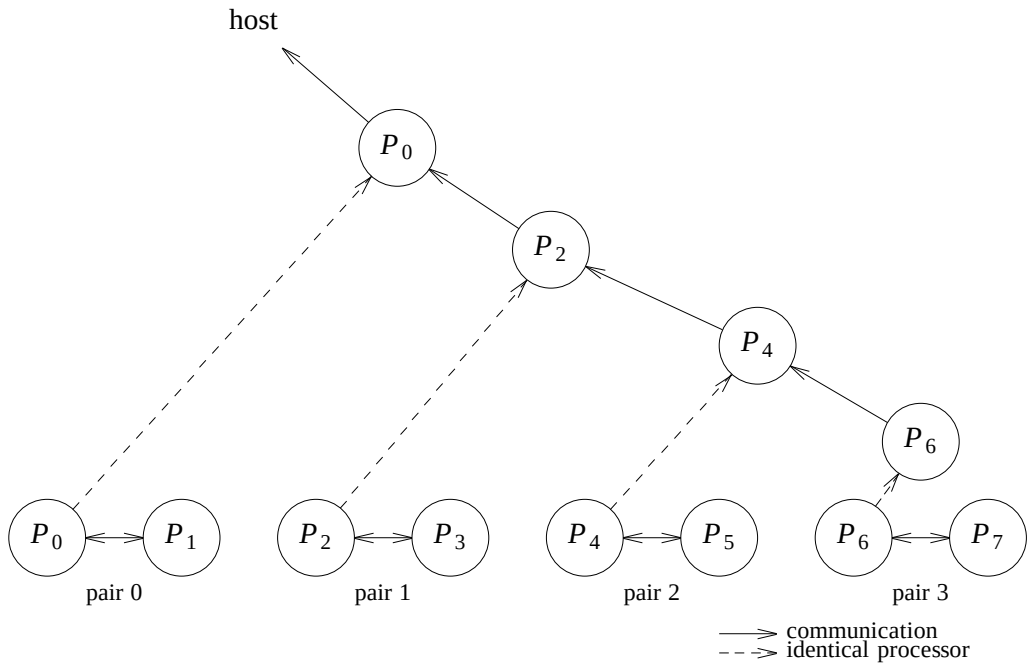


Figure 6.2: Communication path

$$S_p^{balanced} = \frac{n^2 t_s}{t_{balanced}}$$

$$E_p^{balanced} = \frac{1}{\frac{n-1}{n} + \frac{p}{n^2 t_s}\left[(n-1)(t_m+t_c)+\alpha\right]\log p}$$

For constant efficiency,

$$\frac{p}{n^2 t_s}\left[(n-1)(t_m+t_c)+\alpha\right]\log p$$

is required to be constant (assuming $\frac{n-1}{n} \simeq 1$). Hence the isoefficiency is $\Omega(p^2 \log^2 p)$. This is the same as that for the stripes method.

Using the same test set as before, we obtain the efficiencies given in Figure 6.5. The

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.91(2)	0.83(2)				
32	0.90(2)	0.82(2)	0.81(2)			
64	0.94(2)	0.84(2)	0.81(2)	0.90(2)		
128	0.96(2)	0.88(2)	0.82(2)	0.87(2)	1.05(4)	
256	0.97(2)	0.91(2)	0.84(2)	0.83(2)	0.98(4)	1.42(4)
512	0.97(2)	0.94(2)	0.85(8)	0.80(2)	0.91(8)	1.32(4)
1024		0.95(2)	0.88(8)	0.77(16)	0.82(8)	0.96(8)
2048				0.76(16)	0.79(16)	0.90(8)
4096						0.90(16)

Figure 6.3 $t_{squares}/t_{stripes}$

plots of Figures 4.8 - 4.10 are extended in Figures 6.6 - 6.8 to include the speedups for the squares and balanced triangle schemes. For large n, the balanced triangle method is the fastest for small p ($p \leq 16$) and the rectangles method is fastest for large p ($p > 16$).

7. USING THE HOST

One may consider utilizing the processing capabilities of the host to assist in the computation of the connected components. One possibility is to let the host perform the merging step (step 2) of Figure 3.1. The hypercube processors perform step 1 and transmit the spanning structures to the host in packets. A packet is transmitted as soon as it is created. As a result, the host begins merging sooner than step 2 can commence when the stripes or rectangles method is used as in Section 4. Further, the transmission of the spanning structures is overlapped with their computation and merging. For small n and p, we do not expect this utilization of the host to perform better than the raw schemes of Section 4 because of the overhead of communicating with the host and the lack of sufficient merging work. For large n, the merging load is too large for the single host processor to outperform merging by p processors. However, there may be an intermediate range where using the host results in improved performance.

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.50(2)	0.25(2)				
32	0.60(2)	0.34(2)	0.16(2)			
64	0.65(2)	0.44(2)	0.24(2)	0.11(2)		
128	0.70(2)	0.52(2)	0.35(2)	0.18(2)	0.07(4)	
256	0.73(2)	0.59(2)	0.46(2)	0.29(2)	0.13(4)	0.04(4)
512	0.75(2)	0.63(2)	0.55(8)	0.42(2)	0.23(8)	0.08(4)
1024		0.65(2)	0.61(8)	0.56(16)	0.36(8)	0.19(8)
2048				0.65(16)	0.49(16)	0.30(8)
4096						0.39(16)

Figure 6.4 Efficiency of squares method

We experimented with the above scheme using both the stripes and rectangles schemes of Section 4. The results of our experiments are given in Figure 7.1. As is evident, utilizing the host improves performance for n in a suitable range. This range itself changes with p. For larger p (≥ 32 for stripes and ≥ 64 for rectangles) we found no n for which the host could be used in the above manner to improve performance. The optimal packet size to use was found experimentally. This size increases with p.

8. CONCLUSIONS

We have studied several ways to compute the connected components of an undirected graph on a hypercube multicomputer. Starting from the same algorithmic abstraction, one can arrive at programs with different performance depending on the manner in which one partitions and maps the problem, whether or not one attempts to overlap computation and communication, and the attention one pays to load balancing. Of the various methods studied, the balanced triangle scheme of section 6 performed best. Since our programs have good isoefficiency, we expect them to perform well also on hypercubes of much larger size than tested here provided problems of a sufficiently larger size are solved. The required larger size may be predicted using the isoefficiency of the algorithm.

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	0.50	0.23				
32	0.60	0.30	0.15			
64	0.70	0.42	0.22	0.11		
128	0.80	0.56	0.33	0.17	0.09	
256	0.87	0.70	0.48	0.28	0.14	0.07
512	0.94	0.80	0.63	0.42	0.24	0.12
1024		0.88	0.76	0.57	0.37	0.21
2048				0.71	0.52	0.33
4096						0.47

Figure 6.5 Efficiency of balanced triangle method

9. REFERENCES

Carlson, D. 1987. "Fast, Near-Optimal VLSI Algorithm for the Connected Components Problem," IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 12-15.

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

Geist, G.A. and Heath, M.T. 1986. "Matrix factorization on a hypercube multiprocessor," in Hypercube Multiprocessors 1986, ed. M.T. Heath, SIAM, pp. 161-180.

Gopalakrishnan, P.S., Ramakrishnan, I.V., and Kanal, L.N. 1985. "An Efficient Connected Components Algorithm on a Mesh-Connected Computer," Proceedings of 1985 International Conference on Parallel Processing, pp. 711-714.

Gustafson, J. 1988. "Reevaluating Amdahl's Law," CACM, 31, 5 (May) pp. 532-533.

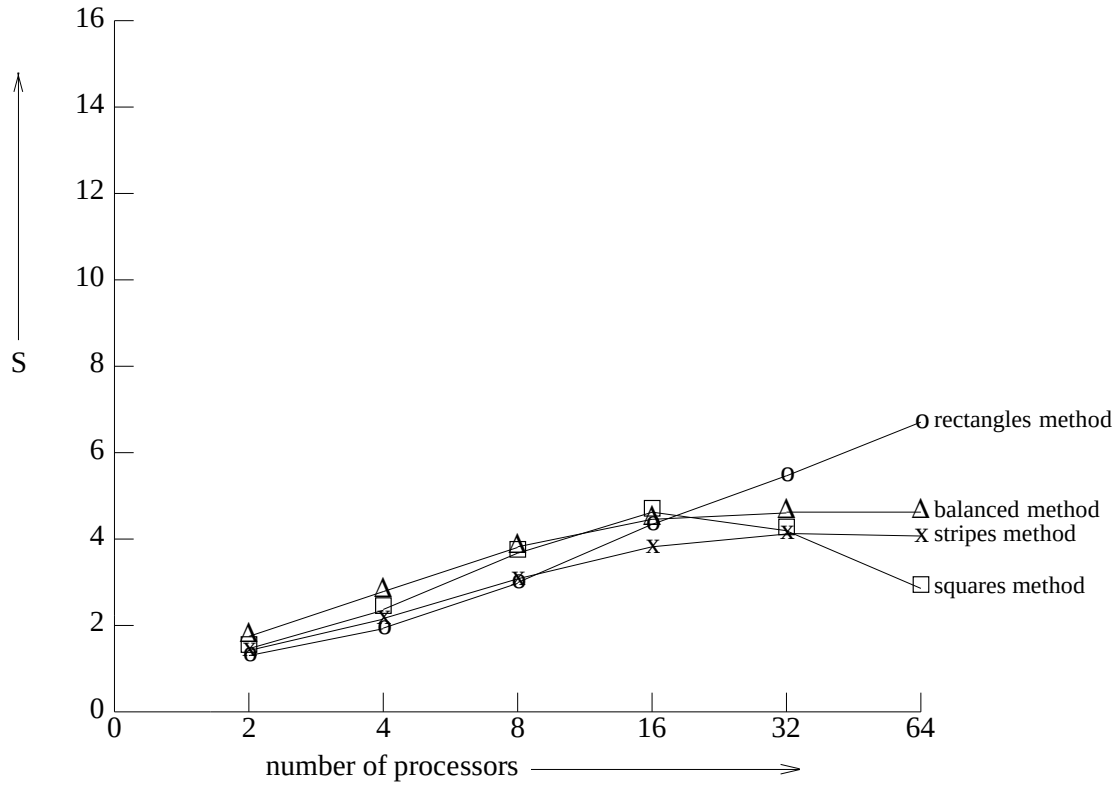


Figure 6.6: $n = 256$

Hirschberg, D.S., Chandra, A.K., and Sarwate, D.V. 1979. "Computing Connected Components on Parallel Computers," Comm. ACM, 22, pp. 461-464.

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

Huang, M. 1985. "Solving Some Graph Problems with Optimal or Near Optimal Speedup on Mesh-of-Trees Network," Proceedings 26th IEEE Symposium on Foundations of Computer Science, pp. 232-240.

Jenq, J. and Sahni, S. 1987. "All Pairs Shortest Paths on A Hypercube Multiprocessor,"

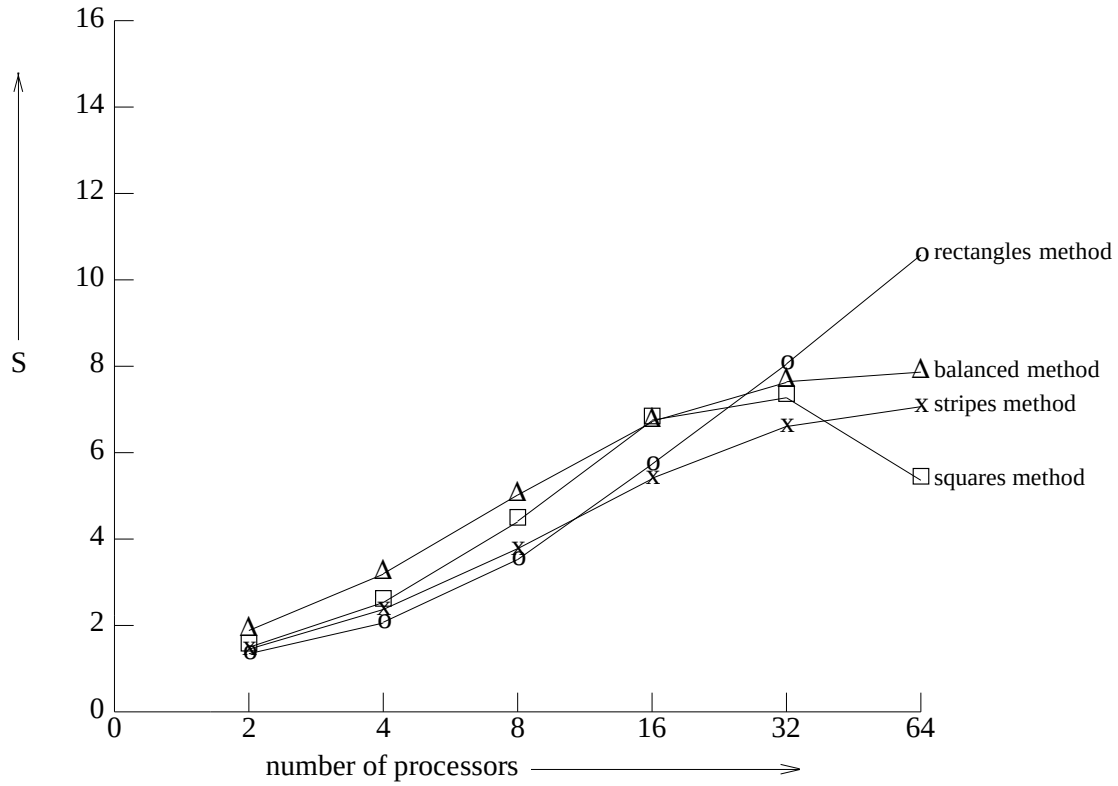


Figure 6.7: $n = 512$

Proceedings of 1987 International Conference on Parallel Processing, pp. 713-716.

Kumar, V., Nageshwara, V., and Ramesh, K. 1988. "Parallel Depth First Search on the Ring Architecture," Proceedings of 1988 International Conference on Parallel Processing, pp. 128-132, Penn State University Press.

Lai, T. and Sahni, S. 1984. "Anomalies in Parallel Branch and Bound Algorithms," Communications of ACM, Vol. 27, pp. 594-602.

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

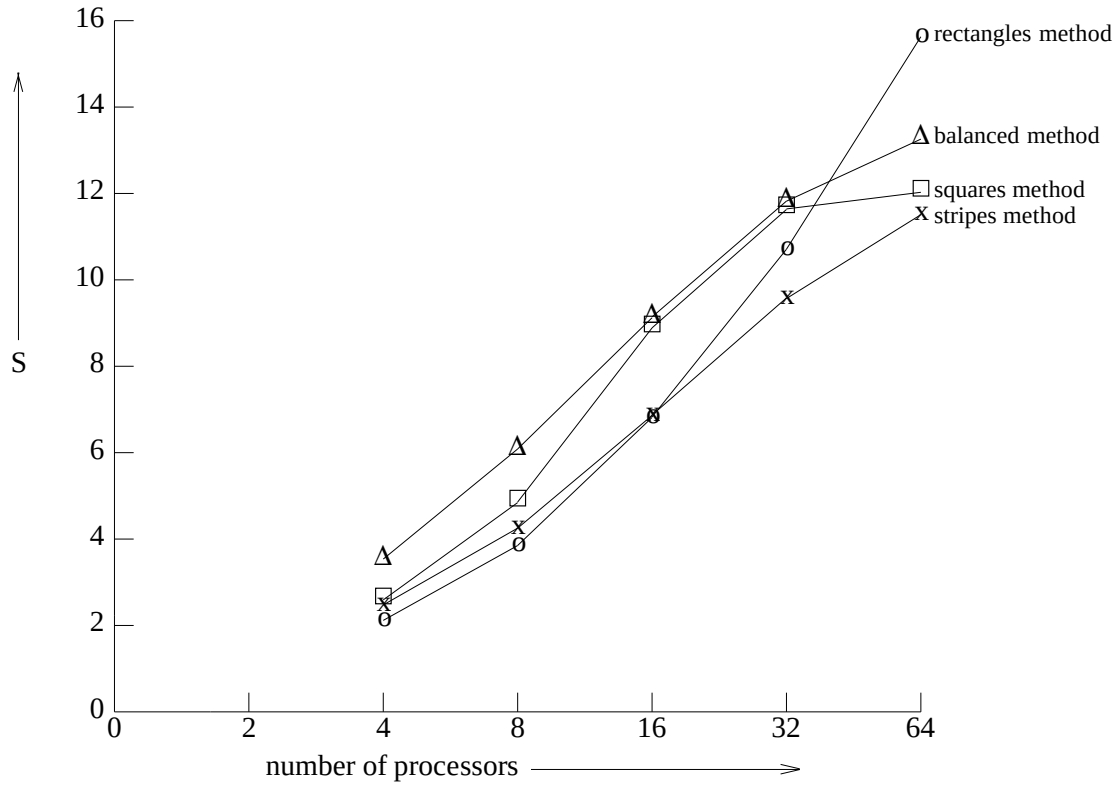


Figure 6.8: $n = 1024$

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

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

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

Ranka, S., Won, Y., and Sahni, S. 1988. "Programming A Hypercube Multicomputer," IEEE Software, Vol. 5, No. 5 (Sep.) pp. 69-77.

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	1.46	1.66				
32	1.21	1.49	2.16			
64	1.11	1.33	1.95	3.35		
128	0.91	0.88	1.51	2.94	5.47	
256	0.94	0.85	0.96	2.27	4.77	9.27
512	0.96	0.90	0.80	1.63	4.03	8.13
1024		0.95	0.87	1.08	2.77	6.74
2048				0.84	1.87	5.23
4096						4.08

(a) $t_{stripes\ using\ host} / t_{stripes}$

size(n)	number of processors(p)					
	2	4	8	16	32	64
16	1.45	1.82				
32	1.21	1.25	1.69			
64	1.09	1.06	1.49	2.32		
128	0.91	0.84	1.25	1.80	3.21	
256	0.93	0.88	0.77	1.28	2.48	4.25
512	0.96	0.93	0.82	0.80	2.00	3.15
1024		0.96	0.90	0.81	1.57	2.49
2048				0.89	0.92	1.95
4096						1.20

(b) $t_{rectangles\ using\ host} / t_{rectangles}$

Figure 7.1

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

Won, Y. and Sahni, S. 1987. "Maze Routing on a Hypercube Multiprocessor Computer," Proceedings of 1987 International Conference on Parallel Processing, pp. 630-637, The Pennsylvania State University Press.