HYPERCUBE COMPUTING: CONNECTED COMPONENTS*

Jinwoon Woo and Sartaj Sahni
University of Minnesota

Abstract

The connected components problem is used as an example to illustrate the programming issues that arise when one uses an MIMD hypercube multicomputer. These issues include selection of algorithmic abstraction, problem partitioning and mapping, overlapping computation and communication, and load balancing.

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

1. INTRODUCTION

The potential of hypercube multicomputers cannot be realized unless one fully understands the process that results in efficient hypercube programs. Programming such a computer 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. The factors we consider in this paper are:

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

These factors are illustrated using the connected components problem. Our model of a hypercube multicomputer is described by the block diagram of 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.

Before describing the multicomputer programming considerations, we describe, in section 2, the various measures used to evaluate multicomputer programs and algorithms.

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 \(t_0\) and \(s_0\), 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 in our hypercube of Figure 1.1) it is often more meaningful to measure the maximum local memory requirement of any node. This is so, 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 \(t_p\) 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 [LAIS84, LIR6], [QUIN86], and [KUMA88], 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 $E_2 = k/4$.

![Figure 2.1](image)

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 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 ineficiency to measure this property. The ineficiency, $ie(p)$, 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, $t_0 = n^3$ 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,

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

where $t_{com}$ represents the time spent in interprocessor communication.

So,

$$S_p = t_0/p - p(t_0/p + t_{com})$$

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 $1/a$. So, $t_0 = work load = cn^3 = opt_{com}$. In other words, the work load must increase at least at the rate opt_{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 $p^{3/2}$ (or more accurately, $(ao/\delta)^{1/2}p^{3/2}$).

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, $ie(p) = \Omega(p)$. Parallel algorithms with smaller $ie(p)$ are more scalable than those with larger $ie(p)$.

3. ALGORITHM SELECTION

As mentioned in [RANK88] 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. [CARL87, GOPA85], [HILZ86], [HUAN88], [NASS80], and [SHIL82] 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 [SHIL82]. 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 $O(n+e)$ processors. This may be run on an $O(n+e)$ node hypercube by using the $O(log^2 n)$ random access read and write algorithms of [NASS81]. The complexity of the resulting hypercube algorithm is $O(log^3 n)$. On a uniprocessor, the connected components can be found in
O(n log n) time, using either depth or breadth first search. For dense graphs, \( e = O(n^2) \) and the speedup, \( S_n = O(n^2/\log n) \). The efficiency \( E_n = O(1/\log n) \). Hence, efficiency declines to zero as \( p \) (and hence \( n \)) increases.

The processor-time product is a measure of the total work (useful and useless) done by a parallel algorithm. The processor-time product of the \( O(n+2e) \) processor hypercube simulation of the algorithm of [SHIL82] is \( O(n+2e \log n) \). For a dense graph, this is \( O(n^2 \log n) \). 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 \( \log n \) more work. Hence, if \( n = 1024 \), then it would take \( \log n = 1000 \) 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 [DEKS81] have developed an \( O(\log n) \) hypercube algorithm to find a spanning forest of an \( n \) vertex graph. This uses \( n/\log n \) processors. This algorithm may be adapted to find connected components in \( O(\log n) \) time. The processor-time product of this adaptation is \( O(n^{3/2}) \). 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 node 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 [HUA85] for the mesh-of-trees multiprocessor. 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 [HORO86]. A depth first traversal could also have been used.

**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;
- for each vertex \( i \) in \( V_r \) do
  - if vertex \( i \) is unmarked then {find a tree for \( i \)}
  - begin
    - add \( i \) to queue and mark it;
    - while queue not empty do
      - if vertex \( j \) in queue then {find a tree for \( j \)}
      - begin
        - delete vertex \( j \) from queue;
        - if \( j \in V_r \) then scan row for vertex \( j \)
          - else if \( j \in V_c \) 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; {of then and for}
- end; {Spanning Forest}

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

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 forest merging. This is done in step 2 of Figure 3.1. 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.

It is readily seen that the total computational work done by the algorithm 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 [SHIL82] and [DEKS81].

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 [JENQ87] 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 node 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 [RAN88]. Along with a data partitioning, one needs to provide a mapping of the data partitions to the node 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 \( a/p \) contiguous
Figure 4.1 shows the partitioning and node mapping for the case \( n = 32 \) and \( p = 8 \). In this figure, \( P_i \) denotes node \( i \) of the hypercube.

To compute the connected components, each node first computes a spanning forest of the given \( n \) vertex graph. This spanning forest is computed using procedure Spanning Forest.

![Figure 4.2: Communication path](image)

![Figure 4.3: Communication path on hypercube](image)

Since a node's adjacency matrix partition has \( n/p \) rows of \( n \) bits each, step 1 takes \( O(n^2/p) \) time. 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 \( V_j-V_i \), scanned in the else clause. So, a total of \( n^2/p + (n-n/p)n/p = 2n^2/p - n^2/p \) accesses to the node's adjacency matrix partition are made. Hence, step 1 takes \( n^2/p(2-1/p)t_c \) time \( (t_c \) is a constant). There are \( \log p \) merge stages with each taking \( (n-1)t_m \) 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 [HOROS86] take slightly more time; linear time can be achieved using the equivalence class algorithm of [HOROS86]). Note that a spanning forest of an \( n \) vertex graph/ubgraph can contain at most \( n-1 \) edges. Each communication of a spanning forest takes at most \( \alpha + (n-1)t_c \) worst case time where \( \alpha \) is the communication startup time and \( t_c \) is a constant. The overall worst case time (excluding the final component identification time) is:

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

Since \( t_m \) is \( n^2/t_c \), the speedup \( S_p^{\text{stripes}} \) is

\[
S_p^{\text{stripes}} = \frac{n^2t_c}{t_{\text{stripes}}} = \frac{n^2t_c}{n^2/p(2-1/p)t_c + (n-1)t_m + (n-1)t_c \log p + \alpha \log p}
\]

and the efficiency \( E_p^{\text{stripes}} \) is

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

For constant efficiency, we require

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

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 \( \Omega(p^2\log p) \). Hence, the inefficiency is \( \Omega(p^2\log p) \). From the equation for \( S_p^{\text{stripes}} \), we get...
\[ S_{\text{stripes}} \leq \frac{n^2 t_e}{2p - 1} \]

Hence, \( E_{\text{stripes}} \leq \frac{p}{2p - 1} \), for graphs which require the examination of all \( n/p \) rows of \( V_r \) and \( n - n/p \) columns of \( V_r^c - V_r \) in step 1. For such graphs, \( E_{\text{stripes}} \leq 2/3 \) for \( p = 2 \), \( 4/7 \) for \( p = 4 \), \( 8/15 \) for \( p = 8 \), etc. Note that on graphs other than these the efficiency can be higher. For sufficiently dense graphs these bounds can be expected to apply.

### 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 \( \frac{n}{2^d} \times \frac{n}{2^d} \) where \( p = 2^d \). Figure 4.4 shows the partitioning and node mapping for the case \( n = 32 \) and \( p = 8 \). The mapping is designed to optimize the spanning forest mergings of step 2. While this partitioning is the same as that used in [JENQ87] for the all pairs shortest paths problem, the mapping to nodes is different. The spanning forest merge order is shown in Figure 4.5. This merge order minimizes the spanning forest size following each merge.

![Figure 4.4: Partitioning into rectangles and processor mapping](image)

![Figure 4.5: Merging sequence of rectangles](image)

The worst case step 1 time for this scheme is \( 2n^2/p t_e \) as in the worst case \( |V_r - V_r^c| = n/p \) and all \( n/p \) rows are scanned in the \textit{then} clause and all \( n/p \) columns in the \textit{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 forest corresponding to an \( i \times j \) partition is at most \( 2l(l \geq 1) \) even though \( l \rightarrow 1 \) is a better bound. When \( d \) is even, at most \( 2 \times \frac{n}{2^{d/2}} \) edges are transmitted by each of the nodes that transmit edges; at most \( 2x \) edges are transmitted in each of the next two stages; at most \( d \) in the next two stages; ...; and at most \( 2^{d/2} \) in the last stage. So, when \( d \) is even, at most
\[
x(1+2+4+8+\cdots+2^{d/2}) = \frac{2^d}{2} - 1 = 2^{d/2}\frac{1}{2} - 1 = 2^{d/2} - 2
\]

edges are transmitted. Note that \( 2^{d/2} = 2n > n-1 \). However a spanning forest 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 forest, 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:
\[
t_{\text{rectangles}} = \frac{n^2 t_e}{p} + \min\{6n, (n-1)\log p\} t_m + \min\{6n, (n-1)\log p\} t_e + \log p
\]

Comparing with \( t_{\text{stripes}} \), we see that the worst case step 1 time for the stripes method is less than that for the rectangles method by \( n^2/p \). The step 2 time for the stripes method is 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:
\[
S_{\text{rectangles}} = \frac{n^2 t_e}{t_{\text{rectangles}}}
\]
\[
E_{\text{rectangles}} = \frac{S_{\text{rectangles}}}{p}
\]
\[
e_{\text{rectangles}} = \frac{1}{2^d} - \frac{2+2^{d/2} - 1}{n^2 t_e} \min\{6n, (n-1)\log p\} (t_m + t_e + \log p)
\]
For constant efficiency,

\[
\frac{E}{n / \log p} \left[ \min \{ 6n, (n-1) \log p \} \left( t_m + t_e \right) \log p \right]
\]

must be constant. When \( \min \{ 6n, (n-1) \log p \} = (n-1) \log p \),

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

\[
\frac{E}{n / \log p} \left[ 6n (t_m + t_e) \log p \right]
\]

must be constant. For \( n \gg \log p \), this requires that \( n \) grow as \( \Omega(p^2) \). 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 \( \Omega(p^2 \log p) \).

From the equation for \( E_{\text{rectangles}} \) we see that for worst case data, \( E_{\text{rectangles}} < \frac{1}{2} \). 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 nodes 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 the 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 \( \frac{2}{3} \) for the stripes method when the test graphs required the examination of all rows of \( V_r \) and all columns of \( V_c \). 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. As \( p \) increases, the efficiency declines because of an increase in the inter node communication overhead. For the rectangles method, again, some efficiencies exceed the 0.5 bound expected.

<table>
<thead>
<tr>
<th>size(n)</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.45</td>
<td>0.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.53</td>
<td>0.28</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.61</td>
<td>0.36</td>
<td>0.20</td>
<td>0.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>0.67</td>
<td>0.46</td>
<td>0.29</td>
<td>0.15</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>0.71</td>
<td>0.54</td>
<td>0.38</td>
<td>0.24</td>
<td>0.13</td>
<td>0.06</td>
</tr>
<tr>
<td>512</td>
<td>0.73</td>
<td>0.59</td>
<td>0.47</td>
<td>0.34</td>
<td>0.21</td>
<td>0.11</td>
</tr>
<tr>
<td>1024</td>
<td>0.76</td>
<td>0.62</td>
<td>0.53</td>
<td>0.43</td>
<td>0.30</td>
<td>0.18</td>
</tr>
<tr>
<td>2048</td>
<td>0.79</td>
<td>0.65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.35</td>
</tr>
</tbody>
</table>

**Figure 4.7: Efficiency of rectangles partitioning**

- o rectangles method
- x stripes method

<table>
<thead>
<tr>
<th>size(n)</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.45</td>
<td>0.25</td>
<td>0.14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.50</td>
<td>0.36</td>
<td>0.20</td>
<td>0.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.57</td>
<td>0.39</td>
<td>0.20</td>
<td>0.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>0.62</td>
<td>0.43</td>
<td>0.23</td>
<td>0.19</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>0.65</td>
<td>0.48</td>
<td>0.27</td>
<td>0.17</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>0.67</td>
<td>0.51</td>
<td>0.44</td>
<td>0.36</td>
<td>0.25</td>
<td>0.17</td>
</tr>
<tr>
<td>1024</td>
<td>0.63</td>
<td>0.48</td>
<td>0.43</td>
<td>0.33</td>
<td>0.24</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>0.65</td>
<td>0.50</td>
<td>0.47</td>
<td>0.40</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>0.65</td>
<td>0.50</td>
<td>0.47</td>
<td>0.40</td>
<td>0.33</td>
<td>0.40</td>
</tr>
</tbody>
</table>

**Figure 4.8: n = 256**

- o rectangles method
- x stripes method

<table>
<thead>
<tr>
<th>size(n)</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>0.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>0.71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>0.73</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>0.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>0.79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.35</td>
</tr>
</tbody>
</table>

**Figure 4.9: n = 512**

- o rectangles method
- x stripes method
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 $n = 256, 512, 1024$ are plotted in Figures 4.8 through 4.10.

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 node $P_0$ of Figure 4.2. We can attempt to reduce the overall time by overlapping data transmission and computation. For this, the odd numbered leaf nodes of Figure 4.2 must transmit their spanning forest edges in packets concurrent with the computation of the spanning forest. If we are sending packets of size $s$ edges, $s < n$, then as soon as $s$ forest 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 forest 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 $(n-1)(s+e)/s$. While this is larger than the $o(n-1)$ 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 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 forest. 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 [WON87] report a 23% reduction in the case of the maze routing problem.

<table>
<thead>
<tr>
<th>size(n)</th>
<th>number of processors(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.49 1.32</td>
</tr>
<tr>
<td>32</td>
<td>1.15 1.18 1.09</td>
</tr>
<tr>
<td>64</td>
<td>0.76 0.92 0.98 1.18</td>
</tr>
<tr>
<td>128</td>
<td>0.46 0.62 0.78 0.84 0.82</td>
</tr>
<tr>
<td>256</td>
<td>0.25 0.38 0.31 0.68 0.73 0.81</td>
</tr>
<tr>
<td>512</td>
<td>-0.17 0.21 0.35 0.46 0.60 0.72</td>
</tr>
<tr>
<td>1024</td>
<td>-0.07 0.15 0.32 0.43 0.61</td>
</tr>
<tr>
<td>2048</td>
<td>-0.19 0.30 0.38</td>
</tr>
<tr>
<td>4096</td>
<td>-0.30</td>
</tr>
</tbody>
</table>

(a) stripes partitioning

<table>
<thead>
<tr>
<th>size(n)</th>
<th>number of processors(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.45 1.64</td>
</tr>
<tr>
<td>32</td>
<td>0.55 0.61</td>
</tr>
<tr>
<td>64</td>
<td>-0.18 0.23 0.26 0.31</td>
</tr>
<tr>
<td>128</td>
<td>-0.63 -0.37 0.10 0.13 0.15</td>
</tr>
<tr>
<td>256</td>
<td>-1.12 -1.15 -0.53 0.00 0.12 0.15</td>
</tr>
<tr>
<td>512</td>
<td>-1.37 -1.43 -1.39 -0.44 -1.49 0.15</td>
</tr>
<tr>
<td>1024</td>
<td>-1.42 -1.40 -0.67 -0.42 -0.14</td>
</tr>
<tr>
<td>2048</td>
<td>-0.95 -0.68 -0.46</td>
</tr>
<tr>
<td>4096</td>
<td>-0.71</td>
</tr>
</tbody>
</table>

(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

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 nodes are assigned the task of finding spanning forests and the others the task of merging spanning forests. Since the strategy of Section 3 transmits spanning forests in packets as these are generated, it is possible for the merge processors to begin their work before the step 1 computation of a spanning forest 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 \( \frac{n}{2} \times \frac{n}{2} \) 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 nodes are grouped into pairs and each node is assigned a row of squares as in Figure 6.1. The nodes begin by computing a spanning forest for their diagonal square. Then the even squares as in Figure 6.1. The nodes begin by computing a spanning forest for their diagonal square. Then the even ones continue to process their squares (only diagonal squares and those to their right are processed) and transmit spanning forests in packets as these are generated, it is possible for the merge processors to begin their work before the step 1 computation of a spanning forest has been completed.

The resulting spanning forest edges to their odd partners. The odd nodes merge spanning forests while the even ones continue to process their squares (only diagonal squares and those to their right are processed) and transmit the resulting spanning forest edges to their odd partners.

This processing of squares consists of the two steps:

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

b) process these spanning forest edges using the union-find algorithm of [HOR086] to eliminate edges that form a cycle when considered in conjunction with those edges already transmitted to the odd partners.

Figure 6.1: Partitioning into squares and processor mapping

Only edges that survive step b) above are actually transmitted to the odd partners. The even nodes do not transmit the spanning forests of their last two squares to their odd partners. On completing all merges, the odd nodes begin to process their squares and transmit spanning forests to their even partners. The even nodes begin to merge after completing the processing of their remaining squares. Once an even node has finished all its work, the resultant spanning forest is transmitted to the even node in the upper adjacent pair (i.e., node \( P_{3i} \) transmits to node \( P_{3(i+1)} \)) for merging. The time, \( t_{\text{square}} \), required by this strategy is given by

\[ t_{\text{square}} = T_0 + T_{\text{wait}} + T_{\text{merge}} \]

where \( T_0 \) is the time taken by \( P_0 \) to finish its work for the pair \( (P_0, P_1) \). \( T_{\text{wait}} \) is the time \( P_0 \) has to wait following \( T_0 \) for the merged data to arrive from \( T_2 \), and \( T_{\text{merge}} \) is the time to merge this data.

Figure 6.2: Communication path

The time, \( t_{\text{square}} \), required by this strategy is given by

\[ t_{\text{square}} = T_0 + T_{\text{wait}} + T_{\text{merge}} \]

where \( T_0 \) is the time taken by \( P_0 \) to finish its work for the pair \( (P_0, P_1) \); \( T_{\text{wait}} \) is the time \( P_0 \) has to wait following \( T_0 \) for the merged data to arrive from \( T_2 \), and \( T_{\text{merge}} \) is the time to merge this data.

Figure 6.3 \( t_{\text{square}}/\text{stripes} \)

Figure 6.4 Efficiency of squares method
The success of this strategy depends considerably on the matching of internode communication times with intra node computation times. Unfortunately, for large \( n \), the time needed to compute the spanning forest of a square is much greater than the time needed to transmit and merge the forest. Hence the merging nodes are often idle. To remedy this load imbalance, the group size may be increased to \( k, k>2 \), nodes. In each group of \( k \) nodes, one node merges while the remaining \( k \) compute spanning forests. This also reduces the rightmost path length of Figure 6.2. Notice that this length is \( p/k \). When \( k>2 \), the merge node of a group only merges forests and the adjacency matrix data for the group is distributed evenly over the remaining \( k-1 \) nodes in the group (again, only data in the upper triangle is needed). Figure 6.3 gives the ratio \( t_{\text{squares}}/t_{\text{stripes}} \) 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. The efficiency table is given in Figure 6.4. Since the spanning forest time increases asymptotically faster than the merge time, for any fixed number \( p \) of processors the ratio \( t_{\text{squares}}/t_{\text{stripes}} \) 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.

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 4.1 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 node initially. This also equalizes the node memory requirements. The resulting efficiencies are given in Figure 6.5 and the plots of Figures 4.8 - 4.10 are extended in Figures 6.6 - 6.8 to include the speeds 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. CONCLUSIONS

We have studied some of the factors that affect the performance of hypercube computer programs. 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. The connected components problem was used as an example to illustrate the effect of these factors.
8. REFERENCES


