Reduction of Communication Delays in Hypercube Programs Based on Run Time Statistics

Scott J. Warren* Joan M. Francioni

Department of Computer Science
Michigan Technological University
Houghton, MI 49931

Abstract. This paper presents a method for reducing communication delays in hypercube programs by attempting to reduce the number of intermediate nodes involved in interprocess communications. In general, the strategy is to observe a program's interprocess communication pattern during a number of executions. When "enough" observations have been made, a new mapping of processes to nodes is created. This new mapping is designed to move processes with high levels of stable interprocess communication as physically close to one another as possible. All further program executions then use the new process-to-node mapping, thereby increasing the overall speed of the program. In this paper, we discuss this method in detail, present results from an implementation on the Intel iPSC/d5, and discuss the overall cost effectiveness of this technique.

1. INTRODUCTION

As with any computer program, the cost of a parallel program involves both design-time costs and run-time costs. The factors which influence this cost for a hypercube program include (1) the design time necessary to make effective use of the hypercube topology, and (2) the communication delays in the running program.

If the topology of a hypercube multiprocessor can be hidden from a program, design time can be reduced considerably. But to design an efficient parallel program for a hypercube, it is necessary to structure the program so that the processes which communicate frequently reside on physically close nodes.

This way the number of message forwardings by intermediate nodes is kept to a minimum, and hence, the communication delays are minimized. For hypercubes which set up a circuit between communicating nodes (for example, Intel's iPSC/2), the closer the nodes are physically, the fewer number of links are necessary for the circuit. This means less time is required for the communication and more links are available for other simultaneous communications. Thus, overall communication delays are minimized in this case as well.

As it turns out, very often the communication behavior of a program is data independent. This means that a program's basic communication patterns can be determined by observing the program during its execution. If statistics are kept during this observation on the frequency of communication between process pairs, then it is possible that a more efficient mapping of processes to nodes can be computed. Further executions of the program using the new process-to-node mapping would then result in a more communication-efficient program.

In this paper, we present a technique called communication redirection (CRD), which does just this. A programmer can design a program with respect to any communication network, and then have CRD assign the processes to the most appropriate nodes of the hypercube. Whether or not faster execution of the program is possible, however, depends on the program's structure. A program written specifically for a hypercube, which uses the most effective mapping of processes to nodes, can not be improved by using CRD. On the other hand, not all programs written for hypercubes do use the most effective mapping. These programs, along with ones not written for a hypercube topology at all, are the ones which can benefit from CRD.

---

This work was assisted by research equipment supported by NSF Grant No. DCR-8404909

* Current address: Hewlett Packard, Fort Collins, Colorado
CRD is designed to be implemented as an operating system service which the programmer is free to use or omit. This makes it easy to observe the communication behavior of the application program during its execution. Also, the operating system can reassign processes to nodes according to the new mapping and redirect all transmissions to their new destinations transparent to the application program. In the rest of this paper, we describe the CRD technique in more detail, discuss the costs involved, and present results obtained from an implementation on the Intel iPSC/d5 hypercube.

2. BACKGROUND

The CRD method consists of three distinct phases: statistics gathering; process-to-node mapping; and communication redirection. During the statistics gathering phase, information is collected as the application program runs under its original processor-to-node mapping. During the process-to-node mapping phase, the application program does not run at all, and the mapping is computed based on the statistics from the previous phase. When the new mapping is determined, the communication redirection phase begins. Application programs which benefit from CRD run in this phase from then on.

The problem of optimally assigning processes to nodes is known to be NP complete [4, 7]. Hence, the problem of effectively mapping processes to nodes is the subject of much research [1, 3, 6, 10]. For this study, however, the emphasis was not on computing this mapping. Rather, the focus was primarily on the concept of communication redirection based on information observed during previous runs.

2.1. Necessary Conditions

The usefulness of this method is dependent on two conditions. For one, it should be applied only to an application program that is to be executed more than \( p \) times, where \( p \) is the break-even point shown in Figure 1. In this graph, line segment A represents the added costs incurred during the statistics gathering phase. During the processor-to-node mapping phase, the application program does not run at all, and so this is all overhead cost as shown by line segment B. It is not until the new process assignment map is available that any gain can be realized. This gain is indicated by line segment C having a lower slope than the original cost line. Given that the new process assignment actually increases the application's speed, a savings occurs when the number of executions reaches the break-even point. Programs executing fewer than the break-even number of times do not benefit from this system. The impact of the various phases, which determine the the actual slopes and the location of the break-even point, depend on the particular system's implementation and the application program's structure.

The second condition necessary for CRD to be useful is that a program's future behavior can be predicted based on the behavior of previous executions. Without consistency between executions, a process-to-node mapping may be created which degrades performance instead of enhancing it. Variations in communication patterns are due to either data dependencies or variations in work assignments. Data dependent communication is determined by a program's branching behavior. It has been shown, however, that many branches in compiled code follow the same path (jump or no jump) with surprising consistency [8]. If a conditional branch instruction follows the same path 90 percent of the time, then any communication controlled by that branch will also behave the same way 90 percent of the time. As for variations in work assignments, these usually come about when a central "work queue" is used to assign work tasks to nodes. Fortunately, almost all hypercube algorithms partition work statically, as do many distributed memory algorithms in general [2, 9]. In these situations interprocess communication is much more regular, allowing better predictions of future behavior to be made.

Situations posing the largest problems are those cases where the observed communication behavior totally misrepresents the typical communication behavior. Programs exhibiting change due to data dependencies or algorithmic design will seldomly display such a dichotomy of communication patterns, however. In most cases, communication pattern changes should appear fairly early during the observation executions. When such changes are noted during statistic gathering, the process-to-node mapping algorithm can consider the variations, guarding against the error of assuming future communication behavior will exactly match the observed behavior.
2.2. Assumptions

Based on the above discussion, it is assumed the application program will be executed some number of times. For programs that are executed only once, or fewer than the minimum number of observation runs, the user would simply choose not to use this technique. For simplicity in the implementation, it was assumed that the application program is structured with only one process per node. One final assumption about the application program is that all processes are uniform, i.e., the same code structure is used in all processes. This is consistent with the structure of hypercube programs in general since the number of nodes tends to be on the order of 10 to 100, making separate programs for each node infeasible. The effect of relaxing these assumptions is discussed in the Section 5.

3. METHODOLOGY

This section describes the three phases of CDR in detail. For each phase, a cost is defined which measures the amount of overhead involved.

3.1. Statistics Gathering

The goal of statistics gathering is to determine the communication frequencies between every pair of processes. To achieve this goal it is necessary to record the identity of the sending and receiving processes on each transmission. A natural way to implement the statistic collecting code is to modify the local copy of the operating system’s communication primitives. It is only necessary to modify the send routine(s) or the receive routine(s), not both. Having the sender record the transmission is a straightforward procedure as the sender probably has access to the receiver’s identity. Creating a composite picture of the average communication pattern requires statistics to be saved between executions. After each observation execution, the statistics will be written to a file. At the end of each execution, the current set of statistics is merged with the cumulative statistics file, producing an updated version of the composite communication pattern.

The statistics which need to be gathered include the number of communications between each process pair, the amount of variation in the communication patterns between executions, and the typical length of the messages. It is convenient if the number of communications is converted to relative frequencies, i.e., percentages, instead of absolute frequencies. The mean relative frequency and the variance of the relative frequencies can then provide adequate measurements of communication pattern consistency. Because relative frequencies are saved for each communicating process pair, the decision to relocate processes can be made on a pair by pair basis, rather than an all or nothing basis.

An important issue of this phase is deciding when to stop gathering statistics. The CRD we implemented uses the approach of observing communication patterns for 10 executions. This makes the decision of when to stop gathering statistics a simple one. Also, it is felt that 10 observation executions do not incur excessive penalties, yet are enough to note most communication pattern variations. More elaborate criteria are certainly possible. One possibility is to stop gathering statistics if the communication frequencies between process pairs meet some form convergence criteria.

The cost of the statistics gathering phase of this work can be divided into two classes: the cost of recording each transmission, \( C_{\text{recording}} \), and the cost of merging the current statistics with the cumulative statistics, \( C_{\text{merge}} \). To record each communication, the operating system needs to maintain a transmission log. In CRD, each sender node maintains a communication Frequency table in memory. This table stores the total number of messages and the average message length for each receiver node. Of the different ways of implementing this table, the primary concern is that it must be possible to access an entry very quickly in order to keep \( C_{\text{recording}} \) low.

The second cost, \( C_{\text{merge}} \), is incurred once per execution and has several components. The cumulative information must be read from the statistics file and written back to the file once it is updated. The statistics gathered during execution must be converted from absolute to relative frequencies. When a program terminates, all of the current execution’s statistics which are distributed among the nodes must be combined into a meaningful form. Finally, some time is required to compute the updated version of the composite statistics file from the old file and the current statistics. On some architectures different parts of \( C_{\text{merge}} \) can be done concurrently, saving time.
The total time for gathering statistics can be represented by the following equation:

\[ C_{\text{stat}} = (\text{No. of messages})(C_{\text{recording}}) + C_{\text{merge}}. \]  

The impact of \( C_{\text{recording}} \) is determined by how many messages must be logged during execution. \( C_{\text{merge}} \), on the other hand, is more dependent on the number of processes in the program than on the number of messages sent.

### 3.2. Process-to-Node Mapping

The second phase of this work is where processes are assigned to nodes. The algorithm responsible for this task uses the communication frequency information collected during the statistics gathering phase along with information about the machine's topology. The goal of the mapping algorithm is to assign processes in a way that reduces the number of message forwardings occurring as a program executes. As was mentioned in Section 2, the main focus of this study is on gathering statistics and redirecting communication. Hence, the mapping algorithm used in this version of CRD is just a simple, yet general purpose, greedy algorithm.

The mapping algorithm employs a communication Cost table that has one entry for each communicating process pair. A process pair's communication cost will primarily be a function of:

1. the number of messages transmitted by the pair;
2. information about the "typical" length of messages between this pair; and
3. the physical distance between the processes of the pair.

The Cost table is constructed from the cumulative statistics file created during the observation executions, and so the separate entries for each pair of the statistics file must be combined into one unit. Given these pairs, the mapping algorithm proceeds as outlined in Figure 2. Let \( N \) represent the number of communicating process pairs. The outermost loop of the algorithm iterates once for each communicating process pair, a total of \( N \) iterations. Within the main loop, the three steps that are not constant time operations (marked with asterisks) each operate in \( O(N) \) time. Thus, the mapping algorithm has a run time cost of \( C_{\text{mapping}} = O(N^2) \). The worst case situation for the mapping algorithm is when a program is designed so that each process communicates with every other process. In this case, \( N = \frac{n(n - 1)}{2} \) pairs, where \( n \) is the number of processes in the program, and \( C_{\text{mapping}} = O(n^4) \). However, many parallel programs are designed so that each process communicates with a fixed number of processes that is much less than \( n \).

### 3.3. Communication Redirection

The last phase of CRD is actually redirecting interprocess communications according to the new process-to-node assignment. The two important goals to be met in this phase are speed and user transparency of the redirection. Implementing communication redirection is done in two steps. First, the Process Assignment table is distributed to all nodes before the application program begins. Then, on every interprocess transmission, the Process Assignment table is first consulted to find the correct location of the receiving process.

Because of the assumption that processes are uniform, physically assigning processes to new sites is unnecessary. It is only necessary to alter processes' identities, and this is handled by using the assignment table. The operating system routine(s) for sending messages are modified to do a table lookup to determine the correct location of the receiving process. However, all dependencies based on node number must reflect the programmer's original process-to-node assignment, not the new assignment. For systems which automatically include the sender's node number as part of a message, this implies the routines for receiving messages must be modified to "unmapped" this number back to the programmer's original process assignment. Otherwise, the receive routine(s) can be used without modification. In addition to the communication primitives, any other routines which supply information about node number will need to use the unmapping lookups.

Let \( C_{\text{dist}} \) be the cost of distributing the process assignment table to all nodes, and \( C_{\text{route}} \) be the cost of re-routing one message. The cost of redirection can then be expressed as

\[ C_{\text{redirection}} = M(C_{\text{route}}) + C_{\text{dist}} \]

where \( M \) is the number of messages in the new process assignment. Since \( M \) includes the number of forwarded messages, it will be less than the number of messages in the old process assignment. If the difference in the new \( M \) and the old \( M \) is large.
enough, the CRD version of the program will run faster than the original version. Otherwise, the overhead of the redirection will dominate the communication cost. Either way, the smaller $C_{\text{redrection}}$ is, the faster the CRD version will run. In terms of memory space, the cost of redirection is small. Each node requires enough memory to store a local copy of the Process Assignment table and the size of the table is $O(n)$.

4. IMPLEMENTATION RESULTS

The software written during this investigation was designed for an Intel iPSC hypercube; the particular system used was an iPSC-d5 (dimension 5) hypercube of 32 nodes. Although CRD is intended to be an operating system service, altering the iPSC's operating system was not feasible here. Therefore, the approach used in this work was to write the statistics gathering and other routines in C and directly add them to the sample programs. This required some modification of the application program but the amount of change was kept to a minimum. The new routines were designed to simulate operating system services as much as possible.

Five programs were used to test statistics gathering, process-to-node mapping and communication redirection. One program was Pic, a Particle-In-Cell code from Los Alamos National Laboratory. Since Pic was designed with communication on a hypercube in mind, the CRD technique cannot improve the original communication patterns. So for test purposes, it was modified to use nonoptimal message routing. The four other sample programs are communication only test programs which implement communication patterns not originally intended for a hypercube machine. Their descriptions follow.

1. **Singles** has two processes, a sender and a receiver. The sender transmits 10,000 messages to the receiver. The processes execute on adjacent nodes.

2. **Corners** has 16 pairs of processes which each exchange 1,000 messages. The communicating pairs are set up to be separated by the maximum possible distance, i.e. 5 links.

3. **Ring** implements a 32 process ring structure where each process exchanges 1,000 messages with its left and right neighbor.

4. **Grid**, implements a 6 by 5 grid of processes where each process exchanges 200 messages with its north, south, east and west neighbors.

In all cases, messages are 1K bytes long. In the following sections, the timings of the individual phases of CRD as applied to these test programs are summarized. Complete raw timing measurements can be found in [11].

4.1. Statistics Gathering Timings

Recall from Equation (1), the two main cost factors of gathering statistics were defined as $C_{\text{recording}}$ and $C_{\text{merge}}$. Table 4-1 summarizes $C_{\text{recording}}$'s impact for all five sample programs. The Singles program was designed to isolate the value of $C_{\text{recording}}$. By having only two processes, a sender and a receiver, delays from contention and interruptions are eliminated. Based on the time to send a 1K message between adjacent nodes (3.5ms), it was determined that $C_{\text{recording}} = 0.0621 \text{ms}$ [5,11]. In some cases, $C_{\text{recording}}$ can be overlapped by useful work, or it occurs when a process would have waited anyway. In these situations $C_{\text{recording}}$'s impact is reduced or eliminated. An interesting case is illustrated with the Ring program where the average execution time with logging is lower than without logging. In this particular case, it is believed that the extra overhead of each transmission makes the processes synchronize better, which in turn offsets the cost of logging messages.

Table 4-2 shows the average $C_{\text{merge}}$ values for all sample programs except Singles. (Since Singles was designed specifically to measure $C_{\text{recording}}$, no $C_{\text{merge}}$ values were recorded.) As mentioned in Section 3.1, $C_{\text{merge}}$ depends on the number of processes in the program. To obtain a more meaningful measurement of $C_{\text{merge}}$, the timings of the second column are divided by the number of processes in the program and then converted into the time needed to send a 1K message.

Table 4-3 shows the overall impact of statistic gathering on all sample programs except Singles. The increase in execution time of all programs is less than 15% and less than 5% in three cases. One reason for Grid's larger delay is due to the way it communicates. Grid uses the iPSC's sendw and recvw primitives which cause execution to be blocked as the operating system services the request. This causes all processes in Grid to execute in a
very synchronous, lock-step fashion. Thus, any delay, even a small one, in a single process slows the entire program in a way that is more exaggerated than the other programs.

4.2. Process-to-Node Mapping Costs

Recall that the communication costs are computed as a function of the number of the messages sent, the average message length and the physical distance between communicating processes. To determine the effectiveness of the mapping algorithm, the communication costs of the Cost table are examined before the mapping algorithm starts and after the algorithm is done. This is done for four programs: Corners, Ring, Grid and Pic. (Singles is excluded because there are only two processes and they are located on adjacent nodes.) The mapping algorithm's results are summarized in Table 4-4. There are two entries for each program. The first entry for each program shows the results obtained with the mapping algorithm's process assignment. The second entry shows the assignment cost when an optimal or near optimal process mapping is made. The second entry represents the best possible results a mapping algorithm could obtain.

The mapping algorithm had its best success with the Corners program. It relocated processes so that all communicating pairs are only 1 link apart, which is the optimal mapping. The cost of an optimal assignment for Ring, having every process pair separated by only 1 link, would be 42% lower than the original cost. With the mapping algorithm, some process pairs were separated by 2 links, resulting in 29% reduction in the communication costs. The mapping algorithm is least effective for the Grid program. While it does remove all pairs that are 5 links apart, it leaves one pair that is 4 links apart and a number of pairs 3 links apart. The cost of the process assignment constructed "by hand", even though possibly not optimal, was much better than the mapping algorithm's cost. As mentioned earlier, Pic's communication pattern was altered to be nonoptimal in this investigation. One reason the mapping algorithm performs better on Pic than on Grid is because Pic's communicating pairs are initially either 1 link apart or 4 links apart. Grid's initial arrangement is less regular.

4.3. Communication Redirection Timings

The overall timing results for \( C_{\text{redirection}} \) are summarized in Tables 4-5 and 4-6. As in Section 4.2, costs here are expressed in terms of the time it takes to transmit a 1K byte message. Column B has values taken when programs are executed in their clean state with nonoptimal communication patterns. Column C has results taken when the program uses an optimal communication pattern without redirection. Column D has results from when the program executes using the process assignment table created by the mapping algorithm to redirect communication. Finally, column E shows the values obtained when a process assignment table constructed "by hand" is used to optimally redirect communication. (In Grid's case the hand mapping is not guaranteed to be optimal, but it is close.) This set of values shows how much of the potential benefit communication redirection can achieve, given a good process assignment.

According to Equation (2), the cost of communication redirection is

\[
C_{\text{redirection}} = M(C_{\text{remote}}) + C_{\text{distribution}} \tag{3}
\]

To distribute the process assignment table, a recursive doubling algorithm was used resulting in

\[
C_{\text{distribution}} = \log_2(n) + 1 \text{ messages.} \tag{4}
\]

Again using the results from the Singles program, it was determined that \( C_{\text{remote}} = .0286 \) messages/redirect [11]. Thus, in the time it takes to send a 1K message between adjacent nodes, \( 1/.0286 = 34.97 \) redirections can be done.

4.4. Analysis of Results

The results in Tables 4-5 and 4-6 show that the CRD technique effectively reduces the average execution time of the sample programs. In some cases the improvement is minimal and in others it is quite extensive. The actual cost effectiveness of the technique, however, depends on the number of times the "new and improved" versions are to be run. For example, the Corners program reaches the break-even point (see Figure 1) the first time redirected communication is used after the 10 statistics gathering runs. On the other hand, the Pic program would have to be run at least 9 additional times under the optimal redirection assignment to reap the benefits of CRD.

5. CONCLUDING REMARKS

Based on this study, it is felt that the CRD tech-
nique can effectively reduce the average execution time of some programs. In addition, the specially designed sample programs show that this method can hide the machine's topology. Gathering statistics during the observation executions does slow down the programs, but the delay is not excessive. The process mapping algorithm developed during this investigation works well enough in some simple cases, but a better algorithm should be designed. The amount of potential gain this technique can achieve is limited by the effectiveness of the mapping algorithm. Because the mapping algorithm is only used one time, a costlier algorithm which produces a more optimal mapping is worth considering.

Although the implementation on the iPSC was done by modifying the application programs directly, it should be fairly easy to incorporate the CRD technique into a machine's operating system. Most of the required steps can be added without changing any of the operating system's routines. In the few cases where a change is needed, such as the communication primitives, the modifications would be minor.

Extending CRD to Distributed Memory Systems

The technique described here was designed specifically for hypercube multiprocessors. One area of future work would be to generalize CRD for any type of distributed memory system. Initially this would involve developing a mapping algorithm to work for irregular topologies. In addition, it would be desirable to relax the assumptions of one process per node and uniform processes on all nodes. Little change is necessary in the statistics gathering or the redirection phases for CRD to handle more than one process per node. The mapping algorithm, however, would have to consider intranode as well as internode communication. Also, the potential benefits of assigning processes of one node to different nodes should be investigated.

To support non-uniform processes on a system of homogeneous nodes, the redirection phase must incorporate the Loader to actually load processes onto different nodes. In a system of heterogeneous nodes, the mapping algorithm must also be modified to handle special process needs resulting from different node structures. In both cases, the statistics gathering code need be no different than for uniform processes.

REFERENCES

Table 4-1. Impact of \( C_{\text{recording}} \).

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Execution Time Without Stats (ms)</th>
<th>Execution Time With Stats (ms)</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singles</td>
<td>27,892.0</td>
<td>28,513.0</td>
<td>2.2</td>
</tr>
<tr>
<td>Corners</td>
<td>96,557.0</td>
<td>97,829.0</td>
<td>1.3</td>
</tr>
<tr>
<td>Ring</td>
<td>81,924.5</td>
<td>81,623.0</td>
<td>-0.4</td>
</tr>
<tr>
<td>Grid</td>
<td>32,502.0</td>
<td>33,619.5</td>
<td>3.4</td>
</tr>
<tr>
<td>Pic</td>
<td>111,591.5</td>
<td>112,208.5</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 4-2. Observed \( C_{\text{merge}} \) values.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>( C_{\text{merge}} ) (ms)</th>
<th>Number of Processes</th>
<th>Adjusted ( C_{\text{merge}} ) (messages/process)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corners</td>
<td>3,317.5</td>
<td>32</td>
<td>29.67</td>
</tr>
<tr>
<td>Ring</td>
<td>3,415.0</td>
<td>32</td>
<td>30.49</td>
</tr>
<tr>
<td>Grid</td>
<td>3,415.0</td>
<td>32</td>
<td>30.49</td>
</tr>
<tr>
<td>Pic</td>
<td>3,415.0</td>
<td>32</td>
<td>30.49</td>
</tr>
</tbody>
</table>

Figure 1. CRD vs. Original Program Cost.

For each communicating process pair {
- Find the most expensive pair;
- Pick one of the two processes to move: mover;
- Maxgain ← 0;
- For each physical neighbor, \( i \), of the fixed process {
  - Compute the cost and benefit of exchanging mover and \( i \);
  - \( \text{Gain} ← \text{benefit} - \text{cost} \);
  - If \( \text{gain} > \text{maxgain} \) then save new maxgain and \( i \);
- If maxgain exceeds 0 then {
  - Switch mover and most beneficial neighbor;
  - Update Cost table for pairs affected by the switch.
}
}

For each communicating process pair {
- Find the most expensive pair;
- Pick one of the two processes to move: mover;
- Maxgain ← 0;
- For each physical neighbor, \( i \), of the fixed process {
  - Compute the cost and benefit of exchanging mover and \( i \);
  - \( \text{Gain} ← \text{benefit} - \text{cost} \);
  - If \( \text{gain} > \text{maxgain} \) then save new maxgain and \( i \);
- If maxgain exceeds 0 then {
  - Switch mover and most beneficial neighbor;
  - Update Cost table for pairs affected by the switch.
}
}

Figure 2. Process Mapping Algorithm.

Table 4-3. Overall Cost of Statistic Gathering.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Cost Before Reassignment (ms)</th>
<th>Cost After Reassignment (ms)</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corners</td>
<td>96,557.0</td>
<td>101,110.0</td>
<td>4.7</td>
</tr>
<tr>
<td>Ring</td>
<td>81,924.5</td>
<td>85,141.5</td>
<td>3.9</td>
</tr>
<tr>
<td>Grid</td>
<td>32,502.0</td>
<td>36,937.0</td>
<td>13.6</td>
</tr>
<tr>
<td>Pic</td>
<td>111,591.5</td>
<td>115,623.5</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Table 4-4. Results of Mapping Algorithm.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Cost Before Reassignment (ms)</th>
<th>Cost After Reassignment (ms)</th>
<th>% Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corners</td>
<td>160,000</td>
<td>32,000</td>
<td>80</td>
</tr>
<tr>
<td>Ring</td>
<td>124,000</td>
<td>88,000</td>
<td>29</td>
</tr>
<tr>
<td>Grid</td>
<td>51,200</td>
<td>44,800</td>
<td>12.5</td>
</tr>
<tr>
<td>Pic</td>
<td>27,984</td>
<td>14,964</td>
<td>46.5</td>
</tr>
</tbody>
</table>

Table 4-5. Communication Redirection Timing Results.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Without Stats or Redirection (ms)</th>
<th>Using Optimal Static Communication (ms)</th>
<th>With Algorithm's Assignment and Redirection (ms)</th>
<th>With Optimal Assignment and Redirection (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singles</td>
<td>27,892.0</td>
<td>27,892.0</td>
<td>28,910.0</td>
<td>28,910.0</td>
</tr>
<tr>
<td>Corners</td>
<td>96,557.0</td>
<td>4,435.5</td>
<td>5,554.0</td>
<td>5,481.0</td>
</tr>
<tr>
<td>Ring</td>
<td>81,924.5</td>
<td>28,564.0</td>
<td>75,109.0</td>
<td>75,755</td>
</tr>
<tr>
<td>Grid</td>
<td>32,502.0</td>
<td>not measured</td>
<td>33,526.0</td>
<td>24,298.5</td>
</tr>
<tr>
<td>Pic</td>
<td>111,591.5</td>
<td>105,036.0</td>
<td>110,772.0</td>
<td>106,735.0</td>
</tr>
</tbody>
</table>

Table 4-6. Percentage of Communication Redirection Improvement.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Without Stats or Redirection (%)</th>
<th>Using Optimal Static Communication (%)</th>
<th>With Algorithm's Assignment and Redirection (%)</th>
<th>With Optimal Assignment and Redirection (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singles</td>
<td>-</td>
<td>0.0</td>
<td>-3.6</td>
<td>-3.6</td>
</tr>
<tr>
<td>Corners</td>
<td>-</td>
<td>95.4</td>
<td>94.3</td>
<td>94.3</td>
</tr>
<tr>
<td>Ring</td>
<td>-</td>
<td>65.0</td>
<td>64.0</td>
<td>64.0</td>
</tr>
<tr>
<td>Grid</td>
<td>-</td>
<td>30.0</td>
<td>25.0</td>
<td>25.0</td>
</tr>
<tr>
<td>Pic</td>
<td>-</td>
<td>5.9</td>
<td>4.4</td>
<td>4.4</td>
</tr>
</tbody>
</table>