Mapping Parallel Computations to Multiprocessor Architectures Considering the Effects of Communication

Loren Schwiebert
D. N. Jayasimha
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277
Email: loren or jayasim@cis.ohio-state.edu

October 15, 1992

Abstract

The problem of mapping parallel algorithms to parallel architectures has been investigated using numerous approaches based on various assumptions. We briefly review some of the previous methods suggested to solve the mapping problem and point out the drawbacks inherent in the past work. We then present the modifications made to correct these shortcomings. Our approach differs from previous approaches primarily by considering the effects of the communication pattern of the parallel algorithm and the communication bandwidth of the parallel architecture. We develop an iterative heuristic algorithm to implement our technique and show that our more realistic method of evaluating the performance of a mapping allows a better mapping to be chosen. Our process uses information about the current mapping to generate an improved mapping. We illustrate the advantage of incorporating this continual refinement. Experimental results are presented to demonstrate the effectiveness of our approach.

Key Words: mapping problem, parallel architectures, directed acyclic task graph, critical edges, communication pattern and communication bandwidth.

1 Introduction

It is widely recognized that the communication overhead of parallel algorithms can have a significant impact on their execution time. Minimizing the impact of communication delays can improve the performance of the parallel algorithm. This is done by using information about the parallel algorithm and parallel architecture to schedule the computation and communication components of the algorithm to minimize the total execution time. The model of the parallel architecture used in this paper is a collection of p processors connected by a network with local memory on each processor (such architectures have traditionally been referred to as distributed memory architectures, examples being hypercube and mesh-connected multiprocessors). The parallel algorithm is modeled as a collection of tasks represented by a directed acyclic graph called the task graph.

The parallel algorithm is first decomposed into tasks. The tasks each require some computation time. They may also require communication with other tasks prior to execution and may transmit results to other tasks after execution. The mapping problem investigates the difficulties of assigning these tasks to the processors. In practice, a parallel architecture is not completely connected. For this reason, the delay associated with communicating between any two processors is not uniform. This fact is exploited when scheduling the algorithm. Finding an optimal mapping is, in general, NP-Complete [3], so work has focused on designing tractable heuristics that generate efficient schedules.

Designing a reasonable heuristic requires an appropriate model of the computation and the communication aspects of both the parallel architecture and the parallel algorithms. As we discuss in the next paragraph, the choice of a goal is also important, since maximizing or minimizing an inappropriate goal may result in an inferior mapping.

In past work, the mapping problem has been examined from different perspectives. Some results have focused on minimizing the communication overhead [1, 6]. However, this does not necessarily result in a good total execution time. Some communication is more crucial to the overall performance of the algorithm. To find the best possible mapping, the arrival time of some messages may have to be delayed so that more critical messages can be expedited. Since the effects of any single communication on the overall algorithm vary, they should not be weighted equally during the mapping. Other unrealistic assumptions made by [1, 6] include requiring as many processors as there are tasks. This lack of generality is not sufficient for most parallel algorithms, which are decomposed into many more tasks than the number of processors in the target machine. They also put unreasonable restrictions on the communication aspects of the architecture or algorithms. For example, [1] assumes that all communications transmit the same amount of data. This assumption is too restrictive for many parallel algorithms. In [1], bandwidth contention is ignored. In [6], contention is considered by explicitly checking the delays caused by the limited bandwidth, but only in the highly restrictive case when all task communication occurs simultaneously. This also does not solve the problem, because contention can arise even when only some of the tasks are communicating at a given time. It must be checked in all cases.

More recent results [2, 5, 7] use the total execution time of the algorithm to guide the search for an optimal mapping. This approach avoids the problem of the previous method, because this approach effectively minimizes the impact of the communication delays, rather than minimizing the total communication delay. This same criterion is also used in this paper. Although a more appropriate goal is used by these papers, various inadequate assumptions are made about the communication aspects of the parallel architecture. For example, in [7], the mapping algorithm assumes that there is infinite bandwidth between the processors. The problem of bandwidth contention is acknowledged in [2, 5], but it is only partially addressed. In the case of [5], the mapping heuristic uses the proximity of the clusters to deal with this problem. This does not solve the problem, because a simple heuristic is unable to determine and hence avoid the contention. The delay caused by bandwidth contention fluctuates during the execution of the algorithm and is difficult to predict. An attempt is made in [2] to measure the contention and adjust the routing of messages accordingly. Unfortunately, they use only a crude estimate of contention. This results in making incorrect decisions. In fact, they show that over a third of their mappings were worse if contention was considered!

Evaluation of these papers and their limitations motivated us to design a mapping heuristic that incorporated more realistic assumptions about both the computation and communication aspects of the parallel algorithm and the communication bandwidth of the parallel architecture into the same model.

2 Definitions

Numerous terms used throughout the paper are defined here.

1. *System Edge* A direct connection in the parallel architecture between any two processors. The connection is assumed to be bi-directional, so a processor can simultaneously send and receive a message along any system edge to which it is connected. It is further assumed that any pair of processors have at most one system edge between them.

2. *Mapping* A mapping is a bijective function from the set of cluster numbers to the set of processor numbers.

3. *Task* A task is a node in the computation graph with a node weight which represents the computation time and edge weights that represent the communication between tasks.

4. *Cluster* A cluster is a totally ordered set of tasks.

5. *Critical Path* This is the longest path in the task graph. It includes both computation and communication. It is possible that there is more than one critical path. The total execution time of a parallel algorithm cannot be reduced unless the length of the critical path is reduced.

6. *Critical Edge* A communication edge is critical if any delay in sending or receiving this message increases the total execution time of the parallel algorithm. The set of critical edges is partially dependent on the cluster assignment.

7. *Critical Task* A task that sends or receives a critical edge.

8. *Critical Cluster* A cluster that contains at least one critical task.

9. *Communication Matrix* This is a $p \times p$ matrix that stores the total amount of communication between any two clusters. When totalling the communication, the direction of the communication between the tasks is not preserved, so $\forall i, j \ni 0 \leq i, j < p, communication(i, j) = communication(j, i)$. The diagonal elements are zero, since the communication between tasks in the same cluster can be replaced with accesses to variables stored in local memory.

10. *Critical Matrix* This is a $p \times p$ matrix that stores the number of critical edges shared between any two clusters. When enumerating the critical edges, no distinction is made between the source and destination tasks, so $\forall i, j \ni 0 \leq i, j < p, critical(i, j) = critical(j, i)$. The diagonal elements are zero, since there are no explicit edges between tasks in the same cluster.

11. *Execution Window* This is the time range in which a task must begin execution. The earliest time in the window is the soonest that the task could conceivably start. The latest time in the window is the latest the task can start without increasing the total execution time of the program.

3 Approach

In order to provide a more comprehensive computation model, we allow the number of tasks in the algorithm to be independent of the number of processors. The execution times of the tasks are variable. The only restrictions placed on each task are that it can begin execution only after it has received all the messages it needs and that it does not send any message until it has completed execution. This is the same computation model used in [7]. All the processors in the architecture are homogeneous.

To approximate the communication properties of the algorithm more closely, the length of any communication message is arbitrary. Store-and-forward packet switching is used. Each message is divided into one or more packets. The length of each message is the number of packets that comprise the message. Message switching requires that an entire message be queued at each intermediate processor between the source and destination processors before any part of the message is forwarded to the next processor. Packet switching requires that only a single packet must be queued at each intermediate processor before it can be forwarded to the next processor. Packet switching reduces the effect of distance on the communication time. This results in lower message latency.

Since all system edges are bi-directional, the bandwidth is limited to the more realistic assumption that a processor can only send and receive one packet at a time on any system edge. This restriction is incorporated into the mapping algorithm so that the effects of bandwidth contention can be measured. The startup costs of the message are assumed to be incorporated into the task execution times, so that a message that is twice as long requires the transmission of twice as many packets. It is assumed that the interconnection network uses a fixed packet length.

Since the mapping algorithm given in [7] and our mapping algorithm both assume clustering as a prior step, simulation results using the two algorithms are compared. Our mapping heuristic has at least two significant differences. Experimental support for the importance of these changes is shown when the results are presented.

1. Our heuristic allows the reassignment of the critical clusters after the initial assignment. This gives our heuristic more flexibility when selecting a mapping.

2. We use a more accurate communication model. This is accomplished by considering the constraints that the bandwidth places on the communication time of each mapping.

Figure 1 illustrates the different ways that bandwidth contention can result in message delay. If this contention is not considered during the evaluation of a mapping, a realistic estimation of the execution time of that mapping cannot be made.

In Figure 1, assume that the clusters have been mapped onto a four processor hypercube. All tasks in a column are in the same cluster. The horizontal axis identifies the processor of the corresponding cluster. The vertical axis gives the time. Each rectangle represents a task with the associated task number inside the rectangle. Each directed edge is a communication message. The weight of each edge indicates the length of the message.

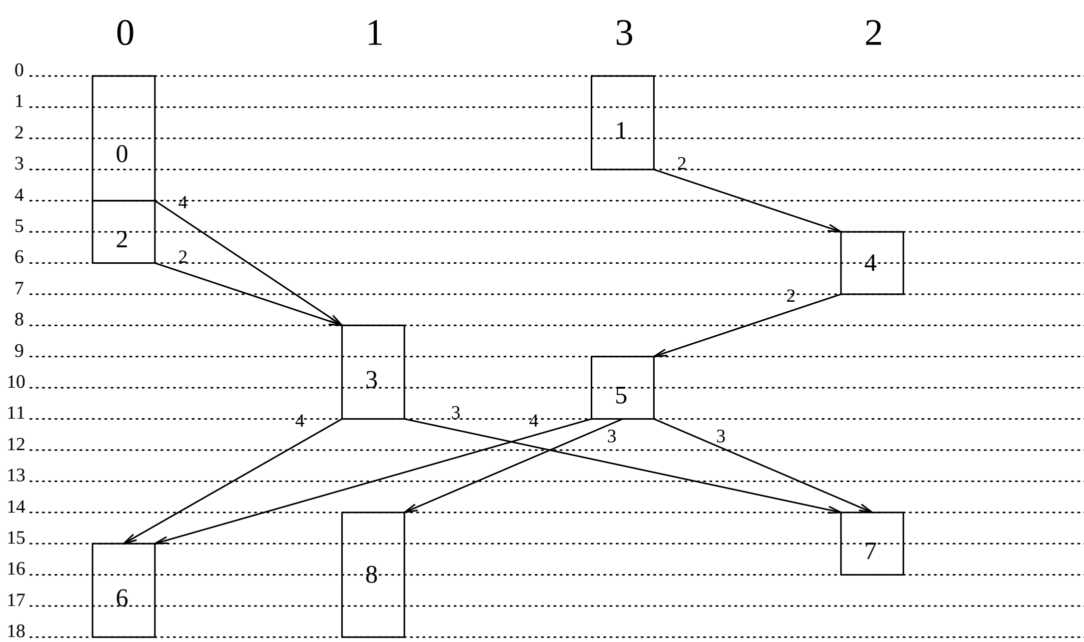

Figure 1: Examples of Bandwidth Related Communication Delays

There are three different situations in which bandwidth contention potentially lengthens the transmission times of messages:

- Finite Bandwidth on any System Edge. Consider the message from Task 0 to Task 3 and the message from Task 2 to Task 3. Both of these require the same system edge. However, Task 0 is still using this system edge when Task 2 is ready to begin transmission. For this reason, it is necessary to delay the sending of the message from Task 2 to Task 3.
- A Limited Number of System Edges at each Processor. There are three messages being transmitted by Task 5. Since this processor has only two system edges, one of the messages must be delayed.
- Limited Paths through the Interconnection Network. At the same time that Task 5 is sending its messages, Task 3 is also sending its messages. The message from Task 3 to Task 6 can begin transmission immediately. However, the message from Task 3 to Task 7 must either wait until the message from Task 3 to Task 6 has completed transmission, or be routed through the processor that executed Task 5. Since Task 5 is currently using all of its outgoing system edges, the message must be delayed in either case.

As previously discussed, a general model requires an arbitrary number of tasks. A common means of handling this situation is via a two step approach. First, a clustering algorithm is used which partitions the tasks into clusters. After clustering, there is exactly one cluster for each processor in the target architecture. The mapping problem is simplified to choosing the appropriate processor for each cluster. The apparent similarity to allowing only as many tasks as processors is only superficial. In reality, the tasks are combined into clusters, but they each maintain their own communication and computational properties. The starting and ending times of each task are still considered individually, as well as the communication of each task. The only constraint the clustering places on the tasks is that a maximum of one task in each cluster can execute at any time. This paper assumes that the tasks have been clustered prior to mapping. Heuristic algorithms for clustering can be found in [4, 5, 8].

The tasks are numbered in increasing order based on their level in the DAG. Although the task graph is acyclic, it is possible that the cluster graph is cyclic. It is assumed that the clustering algorithm together with the dependence requirements of the parallel algorithm determine a total ordering for the tasks in each cluster. This ordering allows for more efficient execution of our algorithm.

Since the mapping chosen is based on the total execution time of the algorithm, it is necessary to define the message lengths by their transmission time. In our model, a time unit is defined as the time required for a single packet to be transmitted along a single system edge. The execution times of the tasks are defined relative to this measure of time. Message lengths are defined by the number of packets contained in the message, therefore the message length is the number of time units required to transmit the entire message along a single system edge.

4 The Algorithm

4.1 Notation

The number of processors in the parallel architecture is p. The number of tasks in the parallel algorithm is n. In general, $n \gg p$. Each message is represented by a directed edge between two tasks in the task graph, so the set of all messages is denoted by e. T is used to denote the total execution time of a mapping. The total execution time is dependent on the mapping, but the other variables are not. The algorithm uses a function, *distance*(), which returns the length of the shortest path between any two processors. When analyzing the time complexity of the mapping algorithm, it is assumed that this function can be computed in constant time.

4.2 Initialization

This section has all the preliminary work for the mapping algorithm. This includes gathering some data to start the iterative process, and calculating the total execution time of the parallel algorithm on the completely connected architecture.

In Figure 2, assume that the clusters have been mapped onto a four processor hypercube. The axes, rectangles, and directed edges are defined as in Figure 1. Reference is made to Figure 2 throughout the presentation of the algorithm.

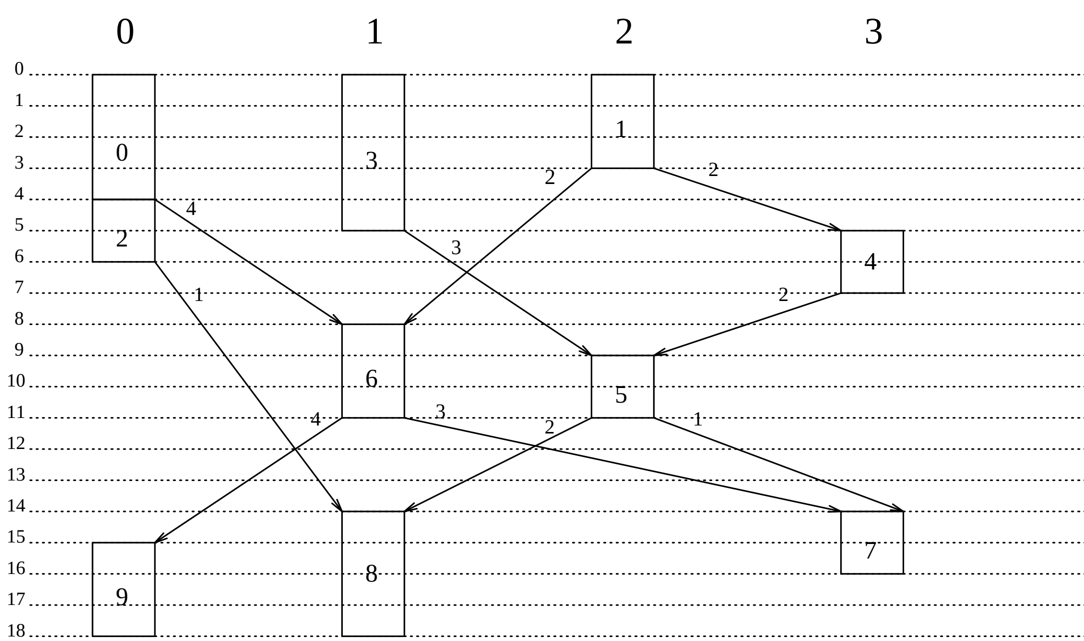

Figure 2: A Sample Task Graph

4.2.1 Calculate the Communication Matrix

This is done by examining each directed edge in the task graph. Each message has a sending task and a receiving task that are assigned to different clusters. Add the length of each message to the corresponding elements of the communication matrix. The direction of the message is ignored. After all the messages have been processed, each element in the matrix corresponds to total message traffic between the two specified clusters. The amount of communication involving each cluster is also totaled. This information is used when mapping the clusters that are not critical. Since all edges must be examined once, and all elements of the communication matrix must be initialized to zero, the time complexity of this step is $O(e + p^2)$.

4.2.2 Create a Mapping for the Completely Connected Architecture

Map the clusters onto the completely connected architecture. This is done using the identity mapping. Since each processor shares a system edge with all other processors, all mappings have the same execution time. The fact that this is a completely connected architecture guarantees that any two clusters share a system edge. This reduces the transmission of all messages to the minimum time required. Since the identity mapping is used, the time complexity of this step is $O(p)$.

4.2.3 Compute the Earliest Starting Times

In order to determine the execution window of each task, it is necessary to compute the earliest starting time of each task. Potential delay due to bandwidth contention is ignored. The delay is determined in a later step.

In this algorithm, we make use of the fact that the tasks have a partial ordering. This allows the algorithm to start with the first task and proceed until the last task is processed. The earliest starting time is the **latest** of these two conditions:

1. All the messages required by this task have arrived. For example, task 6 in Figure 2 cannot start until time 8, because it must wait for the message from task 0.

2. The previous task on this processor has completed execution. This is time 0 for the first task in each cluster. For example, task 2 in Figure 2 cannot start until time 4.

Let i, j, k denote tasks. i is any one of the tasks sending a message to task j. j is the task whose earliest starting time is being computed. k is the previous task in this cluster. Assuming no bandwidth contention, the transmission time of a message using store-and-forward packet switching is:

$$trans(i,j) = messagelength(i,j) + distance(i,j) - 1$$

Using this definition, the earliest starting time of task j is:

$$earliest(j) = \max(earliest(k) + length(k), \max_i(earliest(i) + length(i) + trans(i,j)))$$

For example, the earliest that task 8 in Figure 2 can start is time 14. Even though the message from task 5 to task 8 is only two packets long, $distance(5, 8) = 2$ (because a hypercube with the identity mapping is being used), so the message requires at least three time units to be transmitted.

When each task is evaluated, the messages that it receives are from tasks whose earliest starting time has already been computed. The earliest starting time of the previous task in this cluster is also known. This allows the algorithm to determine the earliest starting time of this task immediately. This must be done once for each task and requires that each communication message be processed once. Therefore, the time complexity of this step is $O(n + e)$.

4.2.4 Compute the Latest Starting Times

To complete the determination of the execution window of each task, it is also necessary to compute the latest starting time of each task. Since we are still interested in achieving the minimum total execution time, the latest starting time chosen must be early enough that it does not increase the total execution time of the algorithm. Potential delay due to bandwidth contention is again ignored. The delay is determined in a later step.

We again make use of the fact that the tasks have a partial ordering. However, in this case it is necessary to start with the last task and work backward until the first task is processed. The latest starting time is the **earliest** of these two conditions:

1. Start early enough that it completes execution in time for the next task on this processor to also meet its deadline. This is the latest starting time of the next task minus the execution time of the current task. For the last task on each processor, this is the total execution time minus the execution time for this task. For example, the latest starting time for task 7 in Figure 2 is at time 16.

2. Start early enough that all the messages it generates arrive by the end of the execution window of the destination tasks. For example, the latest starting time for task 2 in Figure 2 is at time 11.

Again, let i, j, k denote tasks. i is any one of the tasks receiving a message from task j. j is the task whose latest starting time is being computed. k is the next task in this cluster. The transmission time is computed as in the previous step. The latest starting time of task j is:

$$latest(j) = \min(latest(k) - length(j), \min_i(latest(i) - length(j) - trans(j, i)))$$

For example, the latest that task 3 in Figure 2 can start time 0. Even though the message from task 3 to task 5 is only three packets long, $distance(3, 5) = 2$, so the message requires at least four time units to be transmitted.

When each task is evaluated, the messages that it sends are to tasks whose latest starting time has already been computed. The latest starting time of the next task in this cluster is also known. This allows the algorithm to determine the latest starting time of this task immediately. This must be done once for each task and requires that each communication message be processed once. Therefore, the time complexity of this step is also $O(n + e)$.

4.2.5 Evaluate the Bandwidth Contention

For the completely connected architecture, the problem of bandwidth contention is greatly reduced. Figure 1 illustrates the only instance where there could be a bandwidth induced delay even on the completely connected machine. Consider the first example of bandwidth delay. This occurs because the bandwidth between the processors is finite. Regardless of how many system edges there are, this type of delay is unavoidable.

It it necessary to measure the delay experienced due to the bandwidth contention. This allows a more realistic total execution time for the mapping to be found. Since the total execution time of the mapping is more accurately determined, inferior mappings can be correctly rejected. In order to simulate the message transmission, it is necessary to make several assumptions about how the messages are routed through the interconnection network. The assumptions that we have used are given below:

1. Messages are transmitted using store-and-forward packet switching.

2. Each processor has infinite buffer space for messages that it has received and must be forwarded to another processor.

3. Each processor uses a priority queue to store the messages that it injects into the network. The packets are injected as soon as an acceptable system edge is available.

4. The mapping algorithm can determine the order in which the messages generated by a task are placed on its processor queue. The messages from the task are added to the end of the queue. The ordering of the messages is based on the execution window of the receiving tasks. The messages that need to arrive soonest are before the messages that can arrive later.

5. Packets already being transmitted are given priority over packets in the injection queue.

6. Fully-adaptive minimal routing is used to choose the shortest path between the source and destination processors. When multiple routes of the same length are available, a free system edge from one of these shortest routes is arbitrarily chosen. This choice is done independently by each processor along the path from the source to the destination. The completely connected architecture is enhanced to use non-minimal routing and global knowledge of the interconnection network. (Ref. Section 4.2.7)

7. All packets of a message follow the same path through the network. This is the standard method of implementing packet switching.

8. All system edges are bi-directional, so a message can be transmitted in both directions simultaneously.

9. Each packet requires one time unit to travel along one system edge.

These assumptions are made with the intention of approximating an interconnection network closely while maintaining a reasonable execution time for the simulation. This simulation is much more accurate than the method proposed in [2].

We measure the actual contention, rather than trying to minimize it using explicit control over the routing of messages. This allows the bandwidth contention to be approximated based on the routing algorithm. A reasonable modification of these assumptions does not eliminate the problems caused by the bandwidth contention. Regardless of the changes made, bandwidth contention still effects the performance of a mapping. Modifying particular assumptions does not change the fact that this contention must be evaluated to achieve a more efficient mapping.

In order to evaluate the impact of bandwidth contention on the mapping, it is necessary to consider the traffic on the interconnection network at each time interval. Since a packet takes one time unit to travel along one system edge, this is the smallest time interval that can be used. At each time unit it is necessary to do many things. Tasks that can now be started should begin execution. Tasks that have completed execution should transfer the messages to the injection queue. Packets currently being transmitted on a system edge have arrived and must either be forwarded along the next system edge, buffered due to contention, or accepted for a pending task. Packets that are ready to begin transmission should be started if an appropriate system edge is available. If a message must be delayed due to bandwidth contention, this delay is added to the message transmission time and the earliest starting time of the receiving task is adjusted. It is also necessary to adjust the latest starting time of the receiving task if the task is delayed past its latest starting time. This information is used by later stages of the mapping algorithm.

The time complexity of this step is partially dependent on the target parallel architecture. This is because the number of system edges each processor has depends on the topology of the interconnection network. For complexity analysis, it is assumed that this is $\log p$, the actual value for a hypercube. A task can be activated in unit time. Each message must be queued once and each packet must be transmitted once. Other than the message queuing, task completion can be done in unit time. Each packet requires at least one time unit to be transmitted, so the time needed to place the messages on the queue has to be less than the time needed to transmit the messages. All the system edges must be checked at each time unit to determine which are transmitting data. Also, each processor may need to start packets on all of its system edges. All the system edges connected to a processor must be checked to find the one that provides the shortest path to the message destination. The computation of this part dominates the others. The time complexity of this step is $O(Tp\log^2 p)$.

4.2.6 Find the Critical Edges

The critical edges are needed in order to guide the search for an optimal mapping on the target architecture. Since these edges cannot be delayed without increasing the total execution time, preference needs to be given to these messages when the clusters are being assigned. The edges are critical based on the mapping to the completely connected architecture. If it is possible to assign the clusters so that each of these messages is transmitted between directly connected processors in the target architecture, then there is a better chance of producing an optimal mapping. In order to determine the critical edges, it is only necessary to consider edges between tasks that have an execution window with no flexibility (i.e. the earliest starting time and the latest starting time are the same).

Theorem 1 *All critical edges are transmitted between two tasks that have an execution window with no flexibility.*

Proof: Assume that a directed edge between tasks i and j, e_{ij}, is a critical edge. Without loss of generality, assume that task i is the source of the edge and task j is the destination. By definition, any delay in sending or receiving e_{ij} increases the total execution time. In order to minimize the transmission time of e_{ij}, task i must send the message as soon as possible. The soonest the message can be sent is $earliest(i) + length(i)$. Any delay increases the total execution time, so $earliest(i) = latest(i)$. Similarly, task j must receive the message as soon as possible, so $earliest(j) = latest(j)$. Therefore, both task i and task j must have an execution window with no flexibility. □

The reverse is not necessarily true. For example, in Figure 2, the message from task 1 to task 6 is transmitted between two tasks that have an execution window with no flexibility. However, the edge $(1,6)$ is not critical. An edge is critical only if both execution windows have no flexibility and the transmission time is equal to the difference between the ending time of the sending task and the starting time of the receiving task.

The set of critical edges in Figure 2 is $\{(0,6),(1,4),(3,5),(4,5),(5,8),(6,9)\}$. The execution windows for tasks 2 and 7 in Figure 2 are the only ones with any flexibility.

In the worst case, it is possible that all edges are critical. Since all edges must be examined once, and all elements of the critical matrix must be initialized to zero, the time complexity of this step is $O(e + p^2)$.

4.2.7 Calculate a Lower Bound on the Execution Time

The goal of the mapping strategy is to minimize the total execution time of the parallel algorithm. We will prove that within our model, it is impossible to produce a mapping with a total execution time less than that of the completely connected architecture, so this time is a lower bound on the total execution time. It is important to stress that this time may, in fact, not be the optimal execution time. It is possible that the optimal assignment does not achieve this lower bound due to delays caused by bandwidth contention and the fewer system edges on the target architecture. However, finding the optimal execution time is equivalent to finding the optimal mapping. Since this problem is NP-Complete, it is necessary to use this estimation of the lower bound rather than generate the actual optimal execution time.

The two objectives for calculating a lower bound on the total execution time are:

1. To provide a termination condition for the mapping algorithm.

2. To provide a standard of comparison for the results of our mapping algorithm, since finding the optimal execution time is intractable.

Bandwidth contention in the completely connected architecture should be considered when calculating a lower bound on the total execution time. Since the lower bound is used as a termination condition, it is desirable to have as tight a lower bound as possible. Even on the completely connected architecture, there can be bandwidth contention. This contention increases the total execution time. If bandwidth contention is not included in the calculation, it becomes even less likely that *any* mapping on the target architecture can achieve the lower bound.

In order to guarantee that the total execution time remains a lower bound even after including bandwidth contention, the completely connected architecture is not restricted to routing messages along only the shortest path. While a longer path increases the transmission time of a message, the message may still arrive sooner than along a shorter path. This is possible because of the delay that occurs when a shorter path is already in use. For example, in Figure 1, the message from task 2 to task 3 uses the same system edge as the message from task 0 to task 3. The message from task 2 to task 3 may arrive sooner if a longer path is chosen. Even though the delay is minimized, some delay in transmitting this message is unavoidable. In order to determine which choice results in an earlier arrival time, it is necessary for all processors to have global knowledge of the current state of the interconnection network. This is an unrealistic assumption for any real architecture, but is acceptable in this case, because it is used strictly for generating a lower bound. By accounting for this delay when computing the lower bound, a tighter lower bound is possible. All that remains is to prove that this is, in fact, a lower bound.

Theorem 2 *For a given clustering, no mapping on the target architecture can have a total execution time that is less than the total execution time of the identity mapping on the completely connected architecture.*

Proof: First note that the identity permutation mapping is isomorphic to any other permutation mapping on the completely connected architecture. This means that all permutations are equally optimal. We now prove that, given our assumptions, the target architecture cannot provide a better execution time. (The fact that messages must be transmitted as soon as possible and the fact that the tasks have been clustered prevent this mapping from actually being optimal. It is optimal only relative to other mappings that use the same assumptions. This problem is NP-Complete if these two assumptions are removed.)

The messages generated by a task are ordered according to when the destination tasks must begin execution. On the completely connected architecture, the ordering is unimportant if the messages are all destined for distinct processors. None of the messages from a task prevent the transmission of other messages from the same task. In the case where more than one message produced by the same task is destined for the same processor, then one of the messages prevents the other message from obtaining the shortest path. However, since the messages were ordered according to their required arrival time and the tasks in a cluster have a total ordering, the message that needs to arrive first is given the shortest path. The completely connected architecture is able to choose a longer path if the message arrives sooner. This choice is very flexible, in fact, there are $p-2$ paths of length two from which to choose. It is not necessary to consider any longer paths, since all system edges connected to the destination processor lie on one of these paths. Note that since the same thing is done no matter which mapping is used, the identity mapping is still optimal.

This is the optimal execution time on the completely connected architecture. The removal of an edge from the completely connected architecture does not improve the execution time of the parallel algorithm. Each edge that is removed falls into one of two categories. If the edge was not used by the parallel algorithm when executed on the completely connected architecture then its removal does not affect the total execution time. If the edge was used on the completely connected architecture the situation becomes slightly more complex. The messages that used this edge must now be transmitted along another path. This path is longer than the original path, but there are now more shortest paths between the two processors. This does not provide an advantage over the completely connected architecture, however, because it would have only used this shortest path for more than one message if the delay was less than using a longer path. The target architecture must now choose between more than one shortest path of length two or more. The completely connected architecture can choose between one path of length one and $p-2$ paths of length two. Removing an edge from the completely connected architecture does not improve the total execution time of the algorithm. □

The total execution time on the completely connected architecture is the time when the last executing task finishes. Each task must be checked to determine when it finishes, so the time complexity of this step is $O(n)$. As previously mentioned, this time may be less than the optimal mapping on the target architecture can achieve, however, it is used as the lower bound while searching for an optimal mapping.

4.3 Iterative Assignment

4.3.1 Assign the Critical Clusters

If it is possible to achieve the lower bound on the execution time, it can only be done by creating a mapping that approximates the completely connected architecture closely. In order to simulate the completely connected architecture, it is necessary to force the mapping strategy to minimize the transmission time of the critical edges. These edges have a direct bearing on the total execution time. The heuristic used for mapping these critical clusters is to maximize the number of critical edges transmitted between clusters mapped to directly connected processors. Since the critical edges directly affect the total execution time, this increases the probability that the mapping has a shorter total execution time.

This assignment is done after finding out which of the clusters are critical clusters. The total distance of the critical edges is calculated for the current mapping. The algorithm then swaps each critical cluster with every other cluster one at a time and recalculates the total distance. The two clusters are swapped back only if the total distance has increased. The total distance for the initial assignment can be computed in $O(p^2)$ time. In the worst case, all the clusters are critical clusters, so $O(p^2)$ different assignments are tested. Since only two clusters are swapped each time, the total distance for each new assignment can be computed in $O(p)$ time. The time complexity of this step is $O(p^3)$.

4.3.2 Assign the Other Clusters

Normally only some of the clusters are critical clusters. The remaining clusters must also be mapped to minimize the overhead of transmitting their messages. The heuristic used is to map these clusters so the total transmission time of the communication with these clusters is minimized. This is of secondary importance to the critical edge assignment, so these clusters are assigned only to available processors. Packet switching is used and any potential bandwidth contention is ignored during the assignment. The heuristic attempts to minimize the bandwidth contention experienced by these messages by shortening the total distance traveled in the interconnection network.

The assignment of the non-critical clusters is done by repeatedly choosing the cluster with the most total communication. It is placed on the unassigned processor which minimizes the total distance of the messages between this cluster and the clusters that are already assigned. At most, $O(p)$ clusters must be assigned using this method, trying at most $O(p)$ different processors. Checking the total distance requires $O(p)$ time, so the time complexity of this step is $O(p^3)$.

4.3.3 Compute the Earliest Starting Times

This is done using the method presented in Section 4.2.3. The only difference is that the distance function returns the distance between the two processors on the target architecture, rather than the distance on the completely connected architecture. The time complexity of this step is still $O(n + e)$.

4.3.4 Compute the Latest Starting Times

This is done using the method presented in Section 4.2.4. The only difference is that the distance function returns the distance between the two processors on the target architecture, rather than the distance on the completely connected architecture. The time complexity of this step is still $O(n + e)$.

4.3.5 Evaluate the Bandwidth Contention

This is done using the method presented in Section 4.2.5. However, in this case, the delay caused by bandwidth contention occurs more frequently. Any of the situations shown in Figure 1 are possible on the target architecture. Because the likelihood of delay is increased on this architecture, the effect of bandwidth contention increases. Without considering the overhead caused by bandwidth contention, the execution time of the mapping on the target architecture may be underestimated. The time complexity of this step is still $O(Tp \log^2 p)$.

4.3.6 Find the Critical Edges

The critical edges are found using the same algorithm as the one presented in Section 4.2.6. The new set of critical edges is needed to refine the current mapping during the next iteration. The time complexity of this step is still $O(e + p^2)$.

It is important to recognize the relationship between the mapping and the critical edges. The critical edges are critical because of the time it takes to transmit the message from the source task to the destination task. This time is dependent, in part, on the distance between the two processors to which the tasks are assigned. This interdependence is ignored in [7], and so this effectively restricts the ability of their algorithm to generate a satisfactory mapping. Figure 3 presents a trivial example that illustrates this interdependence.

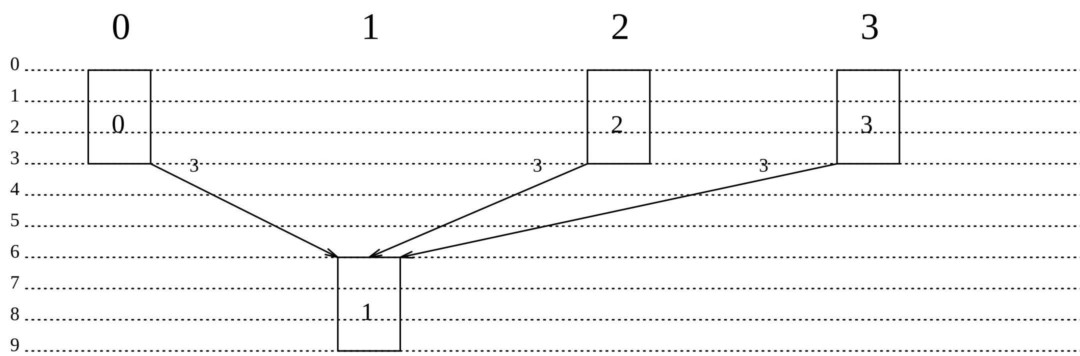

Figure 3: The Set of Critical Edges is Dependent on the Mapping

Assume the clusters have been mapped onto a four processor hypercube. The axes, rectangles, and directed edges are defined as in Figure 1. If the identity mapping is used, then the critical edge is the message from Task 2 to Task 1. If cluster 2 is placed on processor 3 and cluster 3 is placed on processor 2, then the message from Task 2 to Task 1 is no longer a critical edge. For this mapping, the critical edge is from Task 3 to Task 1.

In this trivial example, reassigning the clusters does not change the total execution time. However, in a non-trivial task graph, the change in critical edges can have a substantial impact. Ignoring this interdependence between the cluster assignment and the critical edges impedes the search for better assignments. Essentially, we attempt to minimize the length of the critical path without creating a new critical path. If a new critical path is generated, the mapping algorithm places these clusters closer to reduce that critical path. The best mapping the heuristic finds balances the length of the existing critical paths and the length of any new critical paths that are produced by the mapping. This is the reason that allowing the reassignment of the critical clusters is important. The assignment of the critical clusters may result in a new critical path that cannot be reduced without reassigning some of the critical clusters. By providing more flexibility in the mapping algorithm, a better mapping can be found.

4.3.7 Calculate the Execution Time

As discussed in Section 4.2.7, the total execution time is the time that the last executing task finishes. The time complexity of this step is still $O(n)$.

4.4 Termination

It is necessary to balance the complexity of the algorithm with the need to generate an efficient mapping. In order to maintain a balance between these two goals, three termination conditions are utilized:

- A previous assignment of the clusters to the processors has been repeated. Since the assignment process is deterministic, a cycle of assignments has been found. Continued execution will only repeat previous assignments. The time complexity of checking this condition is $O(p^2)$.
- A mapping with an execution time equivalent to that of the completely connected architecture has been achieved. Since it is impossible to generate a mapping superior to this one, an optimal mapping has been found. No further improvement is possible.
- In order to guarantee an upper bound on the time complexity of the mapping algorithm, it is necessary to provide an additional termination condition. We decided to terminate the mapping algorithm after p different iterations. Based on measurements taken while computing the results presented in Section 5, this seems to give a reasonable trade-off between complexity and performance. Using p iterations was noticeably better than using only $p^{1/2}$ and only slightly worse than using p^2.

4.5 Complexity

There are p iterations of the algorithm in the worst case. Examining each step of the algorithm results in a time complexity of $O(p^4 + np + ep + Tp^2 \log^2 p)$. This can be further simplified by observing that the total execution time of the parallel algorithm, $T \geq n/p$. Similarly, assuming a hypercube architecture, $Tp \log p \geq e$. $T \geq n/p \Rightarrow Tp^2 \log^2 p > np$ and $Tp \log p \geq e \Rightarrow Tp^2 \log^2 p > ep$. The worst-case time complexity of the mapping algorithm is $O(p^4 + Tp^2 \log^2 p)$.

5 Results

The program used to generate these results is written in ANSI C and was run on an HP Snake. All task graphs were randomly generated using task computation times that randomly vary from $1-10$. Similarly, message lengths randomly vary from $1-10$ packets and each task generates from $1-5$ messages. The tasks are uniformly distributed between the clusters. While it would be ideal to generate task graphs from real programs, we feel that the ratios between the computation times, the communication times, and the number of messages generated by the random graphs do reflect the ratios in typical workloads.

Table 1: Relative Execution Times (Optimal = 100)

Problem Size	Ours Modified	Ours	Theirs	Theirs Modified	Random
p = 4, n = 200	113.33	112.09	112.95	113.43	113.24
p = 8, n = 200	116.39	112.28	117.36	117.44	117.47
p = 8, n = 400	117.50	113.79	118.12	117.77	117.60
p = 16, n = 400	120.84	114.05	121.11	118.09	123.33
p = 16, n = 800	121.26	115.21	122.22	118.75	124.40
p = 32, n = 800	122.25	115.57	122.31	120.03	123.84
p = 32, n = 1600	129.05	121.49	131.26	123.04	134.52
p = 64, n = 1600	134.35	123.51	138.51	126.32	142.62
p = 64, n = 3200	143.36	131.57	145.80	133.91	150.05

Table 2: Relative Execution Times Varying the Number of Tasks (Optimal = 100)

Problem Size	Ours Modified	Ours	Theirs	Theirs Modified	Random
p = 16, n = 200	116.12	110.02	117.59	112.29	121.36
p = 16, n = 400	120.84	114.05	121.11	118.09	123.33
p = 16, n = 800	121.26	115.21	122.22	118.75	124.40
p = 16, n = 1600	122.24	117.13	122.94	121.67	124.14
p = 16, n = 3200	119.57	115.51	121.08	119.74	120.99

Five different mapping algorithms were used, and the average results of these algorithms are compared after being run on 100 randomly generated graphs. All tests assume a hypercube architecture. The five algorithms used are:

- **Ours Modified** The mapping algorithm presented in this paper is changed to assume infinite bandwidth. This algorithm is compared with the results produced when bandwidth contention is used to select the best mapping. The best mapping found must then be evaluated while determining bandwidth contention to provide a more realistic assessment of the performance of the chosen mapping. This corrected measure of the total execution time is the time that appears in the results. The reevaluation of the chosen mapping is done only after the mapping algorithm has terminated and only for the sake of comparison.

- **Ours** The mapping algorithm presented in this paper is used to choose a mapping. This allows our mapping algorithm to be compared with all the other mapping algorithms.

Table 3: Relative Execution Times Varying the Number of Processors (Optimal = 100)

Problem Size	Ours Modified	Ours	Theirs	Theirs Modified	Random
p = 4, n = 800	115.26	114.44	115.19	115.10	114.74
p = 8, n = 800	118.04	115.55	118.26	118.37	118.36
p = 16, n = 800	121.26	115.21	122.22	118.75	124.40
p = 32, n = 800	122.25	115.57	122.31	120.03	123.84
p = 64, n = 800	124.95	115.93	128.36	118.82	134.81

- **Theirs** The mapping algorithm presented in [7] is used to choose a mapping. The best mapping found must then be evaluated using packet switching and determining bandwidth contention to provide a more realistic assessment of the performance of the chosen mapping. This corrected measure of the total execution time is the time that appears in the results. The reevaluation of the chosen mapping is done only after the mapping algorithm has terminated and only for the sake of comparison.

- **Theirs Modified** The mapping algorithm presented in [7] is modified to incorporate both packet switching and checking for potential bandwidth contention. This is used to compare our heuristic for cluster assignment with the heuristic given in [7].

- **Random** A random mapping is generated. Packet switching is assumed, and the effects of bandwidth contention are used to find the execution time of this mapping.

Tables 1 – 3 clearly show that a better mapping is chosen by our algorithm when bandwidth contention is considered. Table 3 shows how this difference *increases* with the number of processors. We expect this trend to continue when used for even larger numbers of processors. As seen in Table 2, the relative performance improvement decreases when the number of processors is fixed and the number of tasks is increased. Although the relative performance improvement decreases, the absolute performance improvement increases. Table 1 shows how the relative performance increases with the number of processors if the number of tasks per processor is fixed.

Ignoring the delay caused by bandwidth contention results in choosing a less efficient mapping. For the mapping algorithm given in [7], this relationship holds in all but a few cases. One reason that the difference between the two versions of their algorithm is usually smaller is because the critical clusters cannot be reassigned. Frequently, especially for machines with only a few processors, all the clusters are critical. For those cases, the algorithm given in [7] only tries one mapping!

The reassignment of the critical clusters provides the mapping algorithm with greater flexibility in creating mappings. Since the number of possible mappings is greater, the likelihood of finding a better mapping increases. This is clearly seen by comparing our algorithm with the modified version of the algorithm given in [7]. This relationship does not hold for the mapping algorithms that do not check the bandwidth contention. This is due to the fact that the best mapping is being chosen using inaccurate assumptions.

Except for the smallest problem sizes, the random mapping algorithm has the worst performance. This is not surprising, since the probability that the random mapping generates an optimal mapping is inversely proportional to the number of processors. The variation between the results in this paper is much less than the variation given in [7]. The reason for this relatively limited variation is that packet switching is used. Message switching is used in [7]. Message switching is more sensitive than packet switching to the distance between processors. This results in increased overhead for sending messages between processors that are not directly connected. Since a random mapping is more likely to have critical edges that are not transmitted between directly connected processors, the communication overhead for the random mapping is significantly higher for message switching than packet switching.

Although our algorithm consistently finds a better mapping than the algorithm given in [7], there are infrequent cases when our algorithm does not find a better mapping. We have found two reasons for this. Both heuristics attempt to maximize the number of critical edges between directly connected processors. However, it is possible that many different mappings satisfy this requirement. Because different heuristics are used, it is possible that the heuristic given in [7] chooses a mapping that results in less bandwidth contention. Secondly, a trade-off exists between mapping critical clusters close together to decrease the transmission time of the critical edges, and mapping the critical clusters further apart to allow a larger number of distinct paths between the source and destination processors. When two clusters share a system edge, there is only one shortest path between the two clusters. If more than one message is being transmitted between them simultaneously, one of the messages must be delayed. In this case, it would be better to place the clusters further apart! Since the critical edges tend to be closer with our heuristic, it suffers more from this affect. Note that in both cases the heuristic given in [7] is better due to chance, not some specific aspect of the heuristic.

Other than the random mapping, the greatest difference is consistently between our algorithm and the algorithm given in [7]. This provides additional support for our premises that an accurate model of communication and the continual refinement of the set of critical edges produce a more efficient mapping.

6 Conclusion and Future Work

The mapping strategy presented in this paper demonstrates the need for a realistic model of both the communication requirements of a parallel algorithm and the communication bandwidth of the interconnection network in order to design a heuristic that can produce efficient mappings. Ignoring the communication overhead that occurs due to contention in the interconnection network or ignoring the properties of the message passing facility can result in the rejection of a mapping that is superior to the one that is ultimately chosen. We plan to study what changes can be made to reduce the time complexity of the steps in the algorithm that are computationally intensive while maintaining the better results.

Like all other papers that have addressed this problem, the mappings generated by this strategy use a heuristic. Experimental evidence has been presented to justify the use of this heuristic, however, one aspect we want to examine further is a bound on the ratio of our mapping algorithm versus the optimal. This examination is useful both for quantifying the worst-case and expected results of the mapping algorithm, and for identifying potential refinements to the heuristic.

Many approaches, including our approach, first cluster tasks and then map the clusters. We are convinced that these steps need to be integrated to find a better solution to the general mapping problem. While clustering reduces the complexity of the mapping step, it needs to be made less architecturally independent.

As discussed in Section 5, a trade-off exists between mapping critical clusters close together to decrease the transmission time of the critical edges, and mapping the critical clusters further apart to reduce bandwidth contention. This idea has potential for future research.

Finally, we plan to study the restrictions previously mentioned that each task can begin execution only after it has received all the messages it needs and that it does not send any message until it has completed execution. We plan to consider the ramifications of eliminating this requirement and the complexity of incorporating this change into our model.

References

[1] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, C-30(3):207–214, March 1981.

[2] H. El-Rewini and T. G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, 9:138–153, June 1990.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, NY, 1979.

[4] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. In *ACM International Conference on Supercomputing*, pages 447–450, 1990.

[5] S. J. Kim and J. C. Browne. A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures. In *International Conference on Parallel Processing*, volume 3, pages 1–8, 1988.

[6] S. Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, C-36(4):433–442, April 1987.

[7] J. Yang, L. Bic, and A. Nicolau. A Mapping Strategy for MIMD Computers. In *International Conference on Parallel Processing*, volume 1, pages 102–109, 1991.

[8] T. Yang and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *ACM International Conference on Supercomputing*, pages 633–642, 1991.