AN ABSTRACT OF THE THESIS OF

Chae Young Shin for the degree of Master of Science in Electrical and Computer Engineering presented on August 1, 1994.

Title: A Comparison of Task Scheduling Algorithms on Multicomputers

Abstract Approved: Ben Lee

Dr. Ben Lee

For many years, the von Neumann bottleneck has imposed speed limits on the execution of a program. Because of their sequential nature, von Neumann computers can only execute a single instruction at a time. Instructions that are side-effect free and can be executed in parallel must wait. In an effort to overcome this bottleneck, multicomputers have been developed and implemented. These multicomputers are a new class of computers based on multiple processors. With multiple processors available, instructions can be processed in parallel. However, now interprocessor communication (IPC) delays must be taken into account. A program that is run on several processors may take longer to execute than if it were run on a single processor. The focus of the speed limits has now changed. The new focus now hinges on the efficient partitioning of a program and allocation of those partitions to processors.

Several algorithms have been developed to solve the problem of partitioning and scheduling. Three algorithms were studied under uniform conditions to determine the efficiencies of each. They were Internalization, Balanced Layered Allocation Scheme (BLAS) and Dynamic Level Scheduling (DLS). Simulation studies indicate that BLAS performs the best overall. These algorithms based their communications costs on a simplified IPC cost model. A more realistic message grouping IPC model was developed to test the accuracy of the algorithms which are based on the simplified model. These
simulation studies indicated that the simplified model was a fairly accurate gauge of a more realistic system.
A Comparison of Task Scheduling Algorithms on Multicomputers

by

Chae Young Shin

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Master of Science

Completed August 1, 1994
Commencement June 1995
Approved:

[Signature]
Assistant Professor of Electrical and Computer Engineering in charge of major

[Signature]
Head of the department of Electrical and computer Engineering

[Signature]
Dean of Graduate School

Date thesis is presented: August 1, 1994
Thesis written and typed by: Chae Shin
Acknowledgments

This thesis is dedicated to my parents. Without their help, support and prayers, I would have never continued on to higher education. Their words of encouragement and unwavering support kept me going when I had given up.

My special thanks to Dr. Ben Lee for his guidance and words of wisdom. Not only did he guide me academically, he taught me a little of who I am and of my cultural background that I must not neglect nor forget. Also, special thanks to Dr. Lu who taught me that professors are human too.

I also appreciated the support that my sister Jenell has given me. In her own special way, she visited me regularly and bugged me and helped relieve some of the stress that I was working under.

I also want to extend my gratitude to my friends at Oregon State University. Rita Wells was always there to help guide me through all the red tape associated with school. My friends- Parinda, Ken, Kevin, Manoj, Ravi, Shridar, Mike, Sree, and Shiba- all made my life at school more enjoyable.

I also want to thank Tom, Shridar and Charlie from the ECE support group. Without them I think that I would have broken one of the computers by now.
Table of Contents

CHAPTER 1. INTRODUCTION ................................................................. 1
  1.1 Motivation ................................................................................. 1
  1.2 Partitioning and Scheduling Algorithms ................................. 3
  1.3 Thesis Organization ................................................................. 4

CHAPTER 2. PARTITIONING ALGORITHMS .......................................... 5
  2.1 Program Graphs ...................................................................... 5
  2.2 Balanced Layer Allocation Scheme (BLAS) ......................... 7
  2.3 Dynamic Level Scheduling (DLS) ............................................. 9
  2.4 Internalization ....................................................................... 11

CHAPTER 3. A PERFORMANCE COMPARISON .................................... 14
  3.1 Simulation Environment .......................................................... 14
  3.2 Performance Results and comparisons .................................... 23
    3.2.1 BLAS ............................................................................. 25
    3.2.2 DLS ............................................................................. 26
    3.2.3 Internalization ............................................................... 28

CHAPTER 4. MESSAGE GROUPING ...................................................... 29
  4.1 Why group messages? ............................................................ 29
  4.2 Grouped Messages ................................................................. 30

CHAPTER 5. CONCLUSION ................................................................. 32
  5.1 Review .................................................................................. 32
  5.2 Future Research .................................................................... 33

BIBLIOGRAPHY .............................................................................. 34

APPENDIX ................................................................................... 36

Program Graph Executions ............................................................. 37
LIST OF FIGURES

Figure 2-1. Example of an IF1 program graph with assigned execution times ........ 6
Figure 3-1. Simulator layout ............................................................................ 17
Figure 3-2. L2, Program model on a hypercube .............................................. 18
Figure 3-3. L2, Program graph execution on a mesh ....................................... 19
Figure 3-4. mat10opt, Program graph execution on a hypercube .................... 20
Figure 3-5. mat10opt, Program graph execution on a mesh ............................ 21
Figure 3-6. BLAS, Internalization partitioning comparison .............................. 22
Figure 3-7. Example of thrashing ................................................................. 27
Figure 4-1. Message grouping ....................................................................... 29
Figure 4-2. Message grouping performance ................................................... 31
Figure A-1. 82v, Program graph execution on a hypercube .............................. 37
Figure A-2. 82v, Program graph execution on a mesh .................................... 38
Figure A-3. nwp147, Program graph execution on a hypercube ..................... 39
Figure A-4. nwp147, Program graph execution on a hypercube ..................... 40
Figure A-5. mat5opt, Program graph execution on a hypercube ..................... 41
Figure A-6. mat5opt, Program graph execution on a mesh ............................. 42
LIST OF TABLES

Table 2-1. Example of BLAS using Fig. 2-1 on a 2 processor system ......................... 8
Table 2-2. Example of DLS using Fig. 2-1 on a 2 processor system .......................... 10
Table 2-3. Example of Internalization using Fig. 2-1 on a 2 processor system ......... 13
Table 3-1. List of program graphs ............................................................................ 16
Table 3-2. Maximum speedup with a 2:1 IPC to node execution ratio .................... 24
A Comparison of Task Scheduling Algorithms on Multicomputers

CHAPTER 1. INTRODUCTION

Conventional processors are based on the von Neumann model of computation. The von Neumann model is characterized by a control flow concept which incorporates a single program/data address space and a program counter (PC) which sequences the execution of instructions. However, the simplicity of its design and execution of instructions also has a drawback. Instructions from a program must be processed in a sequential order even if there are sections of code that are side-effect free and can be executed in parallel. This is known as the von Neumann bottleneck [1]. In this model, the execution of a program is limited to the speed of a single processor and its memory. In an effort to overcome this bottleneck, multicomputers have been developed and implemented. "Multicomputers are a new class of computers based on multiple processors" which are connected through an interprocessor communication (IPC) topology [3]. The goal of multicomputers is to alleviate the von Neumann bottleneck by allowing "parallel" sections of code to be run on multiple processors. However, implementation of "parallel" threads of code on multicomputers presents its own problems.

1.1 Motivation

It is the opinion of many computer professionals that parallel architectures are evolving faster than the software to efficiently exploit the available processors. There are two models that have become prevalent in usage for highly parallel architectures. They are the shared-memory model and the message-passing model. The shared memory model is considered among the easiest parallel computer platforms to program since all processors
within the system share a global address space [2]. The drawback to this model is that it lacks scalability for a high number of processors. The message-passing model allows for distributed memories and message-passing across a static IPC network. Due to their distributed nature, multicomputers based on the message-passing model are better in achieving scalable performance [5], but they are harder to program. Multicomputers such as CM-5, Intel Paragon and nCUBE are all MIMD (Multiple Instruction-stream Multiple Data-stream) message-passing computers capable of running multiple threads of code from a single program on different processors. A problem, however, exists with the efficient allocation of these threads across the processors.

This problem of finding the best possible thread allocation has been shown to be an NP-complete problem [8]. However, several algorithms have been developed to combat the time complexity involved in determining a thread allocation. The time complexities of the algorithms that were compared in this thesis were all on the order of $O(N^p)$. These algorithms may produce sub-optimal thread assignments, but they can be completed in polynomial time. These algorithms may be applied in compilers to ease the programmer's burden of thread to processor allocation on message-passing multicomputers.

Several partitioning and scheduling algorithms to solve the allocation problem have been proposed by different authors. Each proclaims that their algorithm performs the best [2][6][12]. These claims may be true for each algorithm tested under their own unique conditions. What is needed is a comparative study over uniform conditions.

Another problem is the accurate modeling of interprocessor communications (IPC) costs in a multicomputer. Most of the algorithms that were researched based their communication costs on a simplified IPC cost model. This issue also needs further study. Are the partitioning and scheduling algorithms still valid when a more realistic IPC model is implemented? Message grouping is proposed as one way to more accurately modeled IPC costs. This was implemented in one of the algorithms for a separate comparison.
Based on the aforementioned discussions, the goal of this thesis is two-fold: the first motivation is to analyze the performance of several different partitioning and scheduling algorithms. The goal is to provide insightful information about the characteristics of the partitioning and scheduling phases of each algorithm. This was accomplished by making comparisons over several different graph types. Simulations were also run under varying IPC conditions. The benefits of such an analysis can be realized when porting serial code to a multicomputer. Serial code can be mapped to a program graph. Depending on the type of the program graph, an appropriate algorithm can be used to assign threads of code to processors within a multicomputer. The second motivation is to study the effects of a more realistic IPC cost model on partitioning and scheduling algorithms.

1.2 Partitioning and Scheduling Algorithms

The problem of mapping threads of code to processors is a two-fold problem. The first problem involves partitioning or clustering a program graph into blocks reduce the program's parallel execution time on an unbounded number of processors. Some algorithms also try to minimize IPC overhead in this stage. In this stage, there are two general types of clustering, linear and non-linear clustering [8]. The second problem runs parallel to the first problem. The scheduling problem involves assigning the partitioned program blocks onto processors. The goals of the scheduling phase are to minimize the IPC costs between blocks of allocated code and ordering the execution of multiple threads on a processor. This phase often increases overall execution time when there are more partitioned blocks than processors.

Three partitioning and scheduling algorithms were used for comparison in this thesis. They are BLAS (Balanced Layer Allocation Scheme) proposed by Lee and Hurson
DLS (Dynamic Level Scheduling) proposed by Sih and Lee [11], and Internalization proposed by Sarkar [2]. Each algorithm is unique in its method of partitioning. BLAS uses a linear clustering method to find an optimal partition, and Internalization uses a non-linear clustering algorithm to find an optimal partition. DLS, on the other hand, does not even utilize a partitioning phase. Each algorithm also has a unique scheduling method. BLAS schedules its linear clusters in the order of decreasing execution time. Internalization uses an iterative test and allocate algorithm for scheduling its clusters. DLS uses a priority-based node scheduling scheme to directly map nodes of a program graph to processors. With these differences in partitioning and scheduling methods, good comparison can be made of the performance of each algorithm in each of its phases. These three algorithms are described in greater detail in the following chapter.

1.3 Thesis Organization

Chapter 2 starts with an explanation of the types of program graphs used by the simulation study. It continues with an explanation of each of the partitioning and scheduling algorithms and any modifications to the original algorithms.

Chapter 3 begins with an outline of the simulation environment for the three algorithms. Simulation results for each of the partitioning and scheduling schemes are then compared and analyzed.

Chapter 4 introduces message grouping in BLAS. The reasons for implementing message grouping are explained. Then simulation results are then analyzed.

Chapter 5 concludes with a review of the results presented in Chapter 3 and the results of message grouping. Suggestions for future research in this area are discussed as well.
CHAPTER 2. PARTITIONING ALGORITHMS

2.1 Program Graphs

The program graphs used in the simulation studies are DAGs (Directed Acyclic Graphs) with an execution time assigned to each node or task. The simulator can run any directed acyclic program graph in the correct file format. The simulator has also been modified to accept SISAL (Streams and Iterations in a Single Assignment Language) program graphs through an intermediate form IF1 (Figure 2.1). The IF1 graphs can be translated into a readable format by the Automator [9]. The edges connecting nodes represent communication between threads of code. They also represent possible IPC delays when the parent and child node of the edge have been allocated to different processors. Note that only the node execution times are pre-assigned. Actual edge IPC costs are not considered until the mapping of the nodes to processors.

A range of program graphs was used in the simulations. The graphs are comprised of hand made test graphs as well as real program graphs. The test program graphs ranged in size from 82 nodes to 192 nodes. The real program graphs were taken from SISAL's front-end which compiles programs into intermediate IF1 graph forms. IF1 graphs are then converted to a readable format by the Automator [9]. They ranged in size from 39 nodes to 134 nodes. The program graphs were comprised of a range of "fat" and "skinny" graphs. The "fat" graphs have an abundant amount of parallelism for the number of nodes that they contained (i.e., maximum parallelism per number of nodes). The "skinny" graphs have a sparse amount of parallelism for the number of nodes within the graph. Later it will be shown that algorithm performance varies widely depending on the amount of parallelism present in the program graph.
All three algorithms were run on the same set of graphs under base conditions. However, it is necessary to note that modifications were made to Internalization's partitioning algorithms in order maintain a uniform base simulation model. Internalization assumes that edge costs are known before the execution of the algorithm. However, this is a catch-22 situation. Accurate edge communication costs are not known until the scheduling phase has completed its assignments of tasks to processors. The resolution to this problem will be answered in each algorithm's description.
2.2 Balanced Layer Allocation Scheme (BLAS)

Balanced Layer Allocation Scheme, proposed by Lee and et al [7], consists of three phases: separation, assignment, and ordering. In the separation phase, serially connected nodes are identified. This phase sorts out the longest path from the root node to the exit node. The longest path is marked the critical path \( N_{cp} \). Unique sub-critical paths called LDPs (Longest Directed Paths) are determined in a recursive manner and are marked \( N_i \), for \( i = 1 \) to \( k \), where \( k \) is the total number of LDPs.

Once the partitioning phase is completed, assignment phase starts. BLAS offers two methods of scheduling. The first method starts by assigning \( N_{cp} \) to an arbitrary processor. Then each LDP \( N_i \) is tentatively assigned to all the available processors. Final assignment of \( N_i \) is determined when all the processors have been tested. \( N_i \) is placed on the processor with the lowest overall execution time. This process is repeated until all \( k \) LDPs have been assigned to processors. The second method only checks and places LDP \( N_i \) in processors that neighbor previously assigned processors (as well at the current processor). The second method is faster since only neighboring processors are checked and assigned to.

The final phase involves ordering of the nodes within the processor to determine the sequence in which graph nodes will be executed. This is necessary since more than one LDP can be assigned to a single processor. Therefore LDPs that are placed on the same processor are reordered. Nodes are shuffled into a new sequence based on when they should execute. Table 2.2 gives an example of the steps involved implementing BLAS on the example program graph in Figure 2.1. An IPC cost of 5 is assumed for this example.
Table 2-1. Example of BLAS using Fig. 2-1 on a 2 processor system.
2.3 Dynamic Level Scheduling (DLS)

Dynamic Level Scheduling is a priority based scheduling algorithm that takes into account the problem of scheduling computation nodes to processors and allocating communication resources to account for IPC costs [12]. It is a simple two phase algorithm that 1) assigns a static priority level to the nodes of a program graph and 2) allocates the nodes directly to available processors based on a dynamically determined priority level.

The first step is to determine the static level of all the nodes of the program graph. This is done by traversing the program graph in reverse order. Static priority levels are determined by a recursive backwards sweep that calculates the longest path from any node in the graph to the exit node. Node to node IPC costs are not included in calculating the static priority level. The next stage involves scheduling ready nodes to available processors. The root node is assigned to a processor. Then the dynamic levels for all nodes that are ready to execute are determined iteratively on each processor. The node/processor pair that has the highest dynamic level is scheduled next.

There are two main factors that determine a node's dynamic priority level: 1) execution and communication time and 2) static priority level. The equation that determines a node's dynamic priority level is as follows:

\[ DL(N_i, P_j, \text{Sum}(t)) = SL(N_i) \ - \ \max \left[ t, DA(N_i, P_j, \text{Sum}(t)) \right] \]

\( SL(N_i) \) is defined to be the static priority level when no communication costs are taken into account.

\( DA(N_i, P_j, \text{Sum}(t)) \) is defined to be the earliest time that all the data for node \( N_i \) is available on processor \( P_j \) at the state \( \text{Sum}(t) \).
Put simply, the dynamic level is the static level of a node minus any IPC cost or delay. Table 2.2 demonstrates how this algorithm works by using the Program graph in Figure 2.1. Again the IPC cost is assumed to be 5.

<table>
<thead>
<tr>
<th>Assign static levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node #</td>
</tr>
<tr>
<td>1. Backwards sweep</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes allocated to processor:</td>
</tr>
<tr>
<td>#0</td>
</tr>
<tr>
<td>2. Allocate the root node</td>
</tr>
<tr>
<td>3. Calculate Dynamic levels of each ready node.</td>
</tr>
<tr>
<td>4. Assign ready node with the highest Dynamic level.</td>
</tr>
<tr>
<td>5. Repeat steps 3 &amp; 4 until all nodes have been allocated.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Dynamic levels of ready nodes on processor #0</td>
</tr>
<tr>
<td>Node #3 = 14</td>
</tr>
<tr>
<td>Node #4 = 17</td>
</tr>
<tr>
<td>Dynamic levels of ready nodes on processor #1</td>
</tr>
<tr>
<td>Node #3 = 9</td>
</tr>
<tr>
<td>Node #4 = 12</td>
</tr>
<tr>
<td>Node #3 = 10</td>
</tr>
<tr>
<td>Node #7 = 13</td>
</tr>
<tr>
<td>Node #8 = 5</td>
</tr>
<tr>
<td>Node #3 = 9</td>
</tr>
<tr>
<td>Node #8 = 0</td>
</tr>
<tr>
<td>etc.</td>
</tr>
</tbody>
</table>

Total execution time : 36

Table 2-2. Example of DLS using Fig. 2 -1 on a 2 processor system.
2.4 Internalization

There are four basic steps in Internalization. They are cost assignment, graph expansion, Internalization and processor assignment [10]. In cost assignment, execution costs are assigned to node and IPC costs are assigned to the edges. Since the IPC costs are not known until actual assignment to processors, a uniform communication cost is assigned to the edges. In graph expansion, a program graph is expanded into an IF1 graph. The goal of this phase is to "scatter" the tasks so that there is a sufficient amount of parallelism exposed for a given number of processors. Small tasks are merged into simple nodes and then mapped onto an IF1 graph. These first two steps are already taken care of since IF1 graphs are a basic part of this simulator.

In step three, internalization determines whether to group nodes into larger processing blocks. The internalization problem is to find a partition that minimizes the critical path length of the task system, i.e. minimizes the completion time of the task system on an unbounded number of processors [10]. For this step, a modified version of Sarkar's algorithm was used. Sarkar's algorithm can be summarized as follows [11]: 1) Sort the edges of the IF1 graph in order of descending order of edge costs. 2) Zero the communication cost of the highest edge of the parallel time does not increase. 3) Repeat step two until all the edges are scanned. Zeroing an edge involves merging the tasks to which the parent and child node belong. Step one presents a problem. It can not be done since all the edge costs have been assigned a uniform cost. The solution that was implemented was to tentatively zero each edge. The edge that would reduce the execution time the most is zeroed. This process is done iteratively until further edge zeroing will increase the overall execution time. This solution is a time intensive process but is necessary to preserve a uniform basis for comparing simulation results.

The processor assignment is also time intensive. In this phase, the blocks which were grouped together in the internalization step are assigned to processors using
calculated IPC costs. A task is iteratively tested on all processors. It is assigned to the processor that reduces execution time. In some cases, there are more tasks than processors. It is then necessary to place tasks on processors that may already have tasks. This allocation often leads to an increase in the execution time. In this case, the task is assigned to the processor that has the lowest overall execution time. This testing and assignment is done to each task. The order of task testing and assignment is random. Table 2.3 gives an example of Internalization using the graph in Figure 2.1. Again there is an IPC cost of 5.
Table 2-3. Example of Internalization using Fig. 2-1 on a 2 processor system. *(parent node, child node)
CHAPTER 3. A PERFORMANCE COMPARISON

3.1 Simulation Environment

One of the main goals in the simulation study was to maintain a uniform condition for running each algorithm. To maintain this uniform condition the following assumptions were made:

(1) All IPC latencies are modeled after a cut-through routing scheme.

(2) For each program graph only the node execution times are initially known.

(3) Execution times for hand made graphs were uniformly distributed over $[t_{min}, t_{max}]$ for the hand drawn graphs.

(1) IPC costs for the program graph edges are modeled after a cut-through routing scheme. This is also known as wormhole routing. Wormhole routing is a message passing model which consists of a large initial communication setup cost, a small hop cost, and a message size. The initial setup cost is the time required to setup a communications channel between the source and destination processors. The message size is associated with the length of the message packet sent. This IPC cost formula is given by the following equation:

$$\text{IPC cost} = t_s + h t_h + m t_w,$$

where $t_s$ is the message startup cost, $h$ is the number of hops, $t_h$ is a cost associated with each hop, $m$ is the message size and $t_w = 1/BW$, where $BW$ is the bandwidth of the IPC.
channel. Edges whose connected nodes are located on the same processor have no IPC cost.

The startup to hop time ratio is based on the nCUBE 3. Its startup latency $t_s$ is assumed to be around 5 or 250 processor cycles and its hop latency $t_h$ is 200 ns or 10 cycles [7]. This startup to hop ratio is 25 to 1 respectively and serves as a reference for $t_s$ and $t_h$ for all simulation runs. The transport time, $ht_h + m_t$, represents the time the message takes to travel from the source to the destination processor. The nCUBE 3 claims to have a BW of 50 Mbytes/s. So, for small messages it is assumed that transport time is dominated by the hop cost $ht_h$ [7].

Different communication to node execution times (C/T) were simulated. C/T ratios were varied to compare the effects that an increasing communication overhead had on the efficiency of the algorithms. The C/T ratios that were used are 2:1, 5:1 and 10:1 respectively.

(2) It was assumed that no initial information was known about the program graph except the node execution times. However, Internalization assumes that accurate IPC costs are known before the partitioning phase. To solve this, a generic IPC cost was assigned to each edge. Modifications were also made to the partitioning phase to account for this generic IPC assignment (chapter 2.4).

(3) A mixture of hand generated test program graphs and graphs converted from real programs were used for the comparison. The real program graphs were taken from SISAL's front-end which compiles programs into intermediate IF1 graph forms. These IF1 graphs were then converted to a readable format by the Automator [9]. Matrix multiplication is one graph that has been converted. The size of the program graphs that were used in the simulation ranged from 39 nodes to 192 nodes. Table 3.1 shows a list of program graphs that were used in the comparison. The maximum amount of parallelism contained in each graph is also listed.
Table 3-1. List of program graphs

<table>
<thead>
<tr>
<th>Graphs</th>
<th># of nodes</th>
<th>Maximum Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>82v</td>
<td>82</td>
<td>7</td>
</tr>
<tr>
<td>L2</td>
<td>192</td>
<td>17</td>
</tr>
<tr>
<td>nwpl47</td>
<td>147</td>
<td>45</td>
</tr>
<tr>
<td>mat5opt</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>mat10opt</td>
<td>134</td>
<td>100</td>
</tr>
</tbody>
</table>

For the hand made graphs, a random execution time was assigned to the graph nodes. For these test program graphs 82v, L2 and nwpl47, execution times were generated in the range of [10,40]. The average execution time of each node was in the neighborhood of 25. The program graphs, mat5opt and mat10opt, are the graphs that are taken from a 5 x 5 and a 10 x 10 matrix multiplication program. The matrix multiplication programs were translated by SISAL and the Automator. The node execution times were preassigned by the Automator and range between 0 and 167 for the 5 x 5 matrix multiplication and between 0 and 250 for the 10 x 10 matrix multiplication graph. The IPC costs are also set accordingly to keep the C/T ratios 2:1, 5:1 and 10:1 for the simulations.

Two IPC topologies were implemented in the simulator. The hypercube and mesh architectures were used for this comparison. These interconnection topologies were chosen because of their scalability and their wide spread use in industry. The mesh that was used is an N by N torus where N is the number processors on a side.

The general design of the basic simulator (Figure 3.1) was kept simple to allow easy modifications or additions. The Simulation Process Control unit receives user input
parameters. They are the $C/T$ ratio used, program graph, the number of processors in the system, the interconnection topology and IPC costs are all requested for a simulation run.

The following graphs (Figures 3.2 - 3.5) are simulation runs of L2 and mat10opt for different $C/T$ ratios and interconnection topologies. These graphs were representative of the other simulation runs. The simulation runs of 82v, nwp147, and mat5opt are listed in the appendix (Figures A.1 - A.6).
Figure 3-2. L2, Program model on a hypercube.
Figure 3-3. L2, Program graph execution on a mesh.
Figure 3-4. mat10opt, Program graph execution on a hypercube.
Figure 3-5. mat10opt, Program graph execution on a mesh.
When IPC costs are removed from the picture and there are as many processors as there are partitioned blocks of code, the scheduling phase becomes redundant because there is no IPC cost to minimize. The efficiency of each algorithm depends on its partitioning phase. The results are discussed in the next sections. Figure 3.6 details the results of BLAS and Internalization on a hypercube. In these simulations IPC costs were zero. DLS is not compared here because it is strictly a scheduling algorithm.

Figure 3-6. BLAS, Internalization partitioning comparison.
3.2 Performance Results and comparisons.

One thing that should be noted is the time complexity of each algorithm. The BLAS algorithm showed an average time complexity of $O((p + V)(N + E))$, for $V << N$ where $p$ is the number of processors, $V$ is the number LDPs and CP (Critical Path), $V$ is the number of nodes in the program graph and $E$ is the number of edges [6]. DLS shows a time complexity of $O(N^2p)$ [13]. In actuality, this algorithm runs faster than it looks. Only the ready nodes are considered for the $N^2p$ when scheduling nodes. This algorithm is the fastest of the three. The time complexity for Internalization using Sarkar's algorithm is of $O(E(N + E))$ [13]. However since it was necessary to modify this algorithm to make it uniform in operational parameters for the comparison, the time complexity ended up being $O(E^2(N + E))$. This algorithm ended up having the worst case time complexity.

From Figures 3.2-3.5, these results are self-evident about the general characteristics of the algorithms. Increasing the $C/T$ ratio has an immediate and uniform effect of reducing the maximum amount of speedup that can be achieved. In essence, the changing $C/T$ can be viewed as changing the granularity of the threads. As the ratio increases, the actual granularity of each thread is decreasing. This suggests that coarse grain threads are better suited to applications in multicomputers.

It's interesting to also note that in all the "fat" and "skinny" program graphs that were simulated, only the amount of parallelism present in the program graph determined the speedup that could be achieved. Speedup $S$ is defined as:

$$S = \frac{T_s}{T_P}$$

where $T_s$ is the sequential execution time of the program graph and $T_P$ is the parallel execution time. Table 3.2 gives an overview of the speedup vs. parallelism for the graphs. The Ratio listed is the maximum parallelism / size of the graph. "Fat" graphs have a ratio
of parallelism of 0.5 or greater. "Skinny" graphs have a parallelism ratio less than 0.5. The speedup $S$ is calculated from the hypercube simulations that had a $C/T$ ratio of 2:1. One would think that there would be greater speedup in a program graph with a higher ratio of parallelism. However, the result from the program graphs nwp147 and mat50pt do not agree. IPC costs act to reduce the amount of parallelism exposed. Larger graphs with the same amount of parallelism have more nodes per branching thread to offset the cost of porting a new thread to another processor. This is all assuming that the granularity of each node is smaller on average than IPC costs.

<table>
<thead>
<tr>
<th>graph</th>
<th>Parallelism</th>
<th>Ratio</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLAS</td>
</tr>
<tr>
<td>82v</td>
<td>7</td>
<td>0.064</td>
<td>2.25</td>
</tr>
<tr>
<td>L2</td>
<td>17</td>
<td>0.088</td>
<td>2.14</td>
</tr>
<tr>
<td>nwp147</td>
<td>45</td>
<td>0.306</td>
<td>4.17</td>
</tr>
<tr>
<td>mat50pt</td>
<td>25</td>
<td>0.641</td>
<td>2.02</td>
</tr>
<tr>
<td>mat100pt</td>
<td>100</td>
<td>0.746</td>
<td>8.31</td>
</tr>
</tbody>
</table>

Table 3-2. Maximum speedup with a 2:1 IPC to node execution ratio.

Figure 3.6 illustrates the performance of BLAS and Internalization on nwp147 and mat100pt when IPC costs are zero. Ideally, the partitioning phase clusters nodes to reduce execution time on an unbounded number of processors. If there is no IPC cost and there are enough processors to place each cluster on its own processor, then the efficiency of the algorithm depends only on the partitioning phase. Figure 3.6 shows that under these conditions BLAS and Internalization perform equally well. However, Figures 3-4 and A-4 indicate that BLAS performs better when IPC costs are present. Therefore, it can be speculated that BLAS's iterative LDP allocation phase or ordering phase is the cause of its
performance over Internalization. In addition, Figure 3-6 also indicates that BLAS performs better than Internalization when there are not enough processors. This also indicates that BLAS's scheduling phase is more efficient than Internalization's.

In these simulations, interconnection topology did not make much difference in the results. This was due to the 25:1 startup time to hop time ratio. This ratio was set up to approximate the startup and delay time associated with an IPC cut-through routing scheme.

3.2.1 BLAS

BLAS performed exceedingly well in the program graphs that were used for the comparison. There are several factors that account for BLAS's performance. In its partitioning phase (i.e. separation phase), it isolates a program graph's critical path and its unique LDPs in a descending order. Internalization partitions program graphs by edge elimination. Edges are tested and eliminated based on whether or not the overall execution time increases or not. The problem that exists with edge zeroing is that it assumes those communication costs between nodes are known before they have been scheduled. This is a catch-22 situation since accurate IPC costs are not known until the nodes are scheduled. Moreover, BLAS schedules (allocation phase) it's partitioned threads in the order of decreasing thread execution time. The critical path is assigned first and then LDPs are assigned to processors. This method is analogous to packing. All the large items are placed in boxes first. The smaller ones are packed last into boxes that still have room. Internalization schedules its blocks of code in a recursive manner. The size of the block that is allocated first is not necessarily largest in execution time. In last phase of BLAS, nodes allocated to processors are arranged into the order of which nodes can start the earliest.
Of the two possible processor allocation schemes that were available only the non-
adjacent processor allocation scheme was used in these comparisons [7]. In the scheduling
phase, LDPs are iteratively tested on processors to determine which processor would
receive the LDP. Adjacent allocation involves testing only the processor most recently
placed and its neighbors to find the next allocation. Non-adjacent allocation tests all
available processors to find the best allocation. Adjacent allocation is quicker than non-
adjacent allocation, however adjacent allocation is also not as efficient.

3.2.2 DLS

The DLS algorithm most often had the worst performance in the comparison. This
poor performance can be accounted to the algorithm's greedy nature. This algorithm,
unlike the other two, has no partitioning phase. The performance of the algorithm is
strictly based on the scheduling phase.

As mentioned earlier, DLS is a priority based algorithm. It takes the highest
dynamic priority level and assigns it directly to a processor to a corresponding processor.
The problem that this algorithm encounters is when a parent node on processor $i$ spawns a
child node to another processor $j$ there is often a return IPC cost which is not taken into
account at the time when the child node is assigned to processor $j$. This also accounts for a
critical fault in the algorithm that seriously degrades the performance of the system when
there are a low number of processors (Figure 3.7). Thrashing occurs if the node that was
previously forked to another processor is needed to join with another node that is on the
parent's processor $i$. The parent processor $i$ has to wait for that extra communication cost.
As a result, this extra communication causes waiting nodes to be misassigned to processor
$j$. This thrashing accounted for the above serial execution performance in the hypercube
and mesh comparisons when the number of processors was limited to two or four. When
more processors exist the nodes waiting for processor $i$ can be assigned to another processor and the problem is alleviated somewhat.

![Figure 3-7. Example of thrashing.](image)

Since this algorithm ignores the possible return IPC cost, DLS's performance suffers when the IPC cost to node execution ratio is high. There was one case (Figure A.4) in which even increasing the number of available processors did not reduce the execution time below serial execution time. Looking back at the program graph, it's shown that there are many micro-synchronization nodes. These are child nodes with several parents. When studying the node/processor allocation for this simulation, it can be seen that the parent nodes of these micro-synchronization nodes have been scattered to different processors and all the graph nodes are clustered around a few processors. From these findings, it can be postulated that these micro-synchronizations are causing DLS's thrashing to occur between the parents and child nodes. The high IPC cost to node execution ratio merely compounded this problem.
It is noted that when this algorithm is allowed a sufficient number of processors and the \( C/T \) ratio is low enough, it has performed better than the other algorithms (figure A.5). The main advantage of this algorithm is that it is considerably faster than others.

### 3.2.3 Internalization

The Internalization performed well compared to DLS. In many cases its performance was close to BLAS. However, due to the modification in the partitioning phase, this Internalization always took the longest to complete. Internalization partitions nodes into blocks by testing edges to be zeroed. The order in which they are tested is random. As noted above in section 3.2.1, this is not always optimal. Accurate IPC costs are not known until scheduling. However, partitioning is based on IPC costs. This problem leads to a sub-optimal partition. In addition, edges are zeroed based on whether the zeroing will reduce overall execution. This edge zeroing does not guarantee that parallel tasks will not be grouped into the same block. The modification that was added is iterative in nature. All the edges are tested before a single edge is zeroed. After zeroing, all the remaining edges are tested again. This procedure is costly but produces good sub-optimal block partition.

Internalization's scheduling phase is an iterative test and allocate loop. Blocks are tested on different processors and placed the processor with the lowest overall execution time. The faults of this scheduling algorithm are noted in section 3.2.1.
4.1 Why group messages?

The current IPC modeling technique used in these simulations as well as ones in the literature assume that different messages originating from a single node on one processor can be broadcast to all children nodes on different processors at the same time. This is an inaccurate model. Transmission of different messages cannot always be run concurrently. To rectify part of this situation, grouping of messages is proposed (figure 4.1). When multiple messages from one node on a processor are passed to multiple nodes on another processor, the messages are grouped together. Grouping messages will increase communication time in the current algorithms, since they overlap their message passing currently. In reality, this is a more accurate representation of modeling the IPC network.

Figure 4-1. Message grouping.
4.2 Grouped Messages

Message grouping was only implemented on BLAS. The following assumptions were made for this study:

1. Only message sizes are used in determining IPC costs.
2. If the parent node is broadcasting (i.e., has more than 4 children nodes), the message costs are not grouped.
3. Execution times for hand made graphs were uniformly distributed over $[t_{\text{min}}, t_{\text{max}}]$ for the hand drawn graphs.

In this study, message size was the only IPC parameter used. The startup and hop costs which usually dominate IPC costs were neglected to magnify the effect that message grouping had on the simulations.

The messages from broadcasting nodes were not grouped since the same message is passed to all the children nodes. The execution times for the nodes were between $[10,40]$ for the hand drawn graphs.

Figure 4.2 shows a simulation run of nwp147. This was characteristic of all the others. The simulations were run on a mesh using a message size to node execution cost of 2 to 1.
As expected, message grouping did increase the overall execution time. However, the difference wasn't as great as it was expected to be. This can be attributed to some nodes being broadcast rather than grouped.
CHAPTER 5. CONCLUSION

5.1 Review

In this thesis, three partitioning and scheduling algorithms were compared against one another. A description and analysis of each of the algorithms were discussed. Two of the algorithms, BLAS and Internalization, performed noticeably better in most cases. Their partitioning algorithms were not designed around a prioritization of graph nodes but rather around grouping of sequential threads of code. BLAS achieves this by finding LDPs of a program graph. Internalization partitions by iteratively testing and zeroing program graph edges. DLS did relatively worse. Although faster in general, this greedy algorithm lacked foresight to determine if a particular allocation would incur extra IPC overhead and waiting delays later on in the program graph.

The simulations with lower $C/T$ ratio performed the best. The speedup ratio was higher for these cases. This seems to indicate that graphs with larger granularities are more efficient in a multicomputer environment.

The results from message grouping were as expected. The overall execution time increased but not as much as expected.

This thesis provided a comprehensive comparison of the performances of several partitioning and allocation algorithms. Each algorithm was individually analyzed against its performance on a variety of fat and skinny graphs and against other algorithms.

In addition, the code for the Internalization and DLS algorithms and the mesh IPC model were written and implemented by Chae Shin.
5.2 Future Research

Although the research done in this thesis is indicative of performances of a variety of program graphs, further research can be done on a larger variety of program graphs. In addition, the basic simulation module was set up to run either a single algorithm or a set of algorithms. Further algorithm models can be added to give a greater range of comparisons and a further understanding of graph allocation algorithms.

In further studies, larger message sizes can be incorporated into the IPC costs. Although startup time still dominates the IPC cost, larger message sizes may effect allocation in the scheduling phase.

More accurate IPC models like message grouping can be incorporated to verify that these algorithms are still valid when realistic IPC costs are applied. In addition, more IPC models can be added. This thesis used cut-through routing as the basis for calculating IPC costs. A store and forward model can be implemented.
BIBLIOGRAPHY


APPENDIX
Program Graph Executions

Figure A-1. 82v, Program graph execution on a hypercube.
Figure A-2. 82v, Program graph execution on a mesh.
Figure A-3. nwp147, Program graph execution on a hypercube.
Figure A-4. nwp147, Program graph execution on a hypercube.
Figure A-5. mat5opt, Program graph execution on a hypercube.
Figure A-6. mat5opt, Program graph execution on a mesh.