Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs

Daniel W. Watson † John K. Antonio †† Howard Jay Siegel †† Mikhail J. Atallah †††

†Department of Computer Science
Utah State University
Logan, UT 84321-4205

††Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285

†††Computer Sciences Department
Purdue University
West Lafayette, IN 47907-1398

Abstract

The problem of minimizing the execution time of programs within a heterogeneous environment is considered. Different computational characteristics within a parallel algorithm may make switching execution from one machine to another beneficial; however, the cost of switching between machines during the execution of a program must be considered. This cost is not constant, but depends on data transfers needed as a result of the move. Therefore, determining a minimum-cost assignment of machines to program segments is not straightforward. A previously presented Block-Based Mode Selection (BBMS) approach is used as a basis to develop a heuristic method for assigning machines to program segments of data-parallel algorithms. Simulation results of parallel program behavior using the heuristic indicate that good assignments are possible without resorting to exhaustive search techniques.

1. Introduction

One of the benefits of heterogeneous parallel processing is that programs can be executed on those machines that best exploit the parallelism in each part of the program. One of the associated challenges of implementing heterogeneous systems is finding a mapping of program segments to parallel machines. In [15], a Block-Based Mode Selection (BBMS) framework for estimating the relative execution time of a data-parallel algorithm in an environment capable of the SIMD and SPMD (Single Program - Multiple Data) modes of computation was presented, where SPMD is MIMD with the restriction that all PEs are executing copies of the same program. The BBMS framework provides a methodology whereby static source-code based decisions of computational mode of execution can be made for each portion of an algorithm. One of the assumptions of the framework in [15] is that the cost of performing mode changes in a heterogeneous system is constant.

In the model of heterogeneous computing assumed here, each segment in a program has a set of zero or more data elements that must be available to it (i.e., on the same machine where the program segment is to be executed). Because each data element may be used and/or generated by other program segments, and because program segments may execute on different machines within the system, it may be necessary to transfer data elements needed by a segment before execution can begin. The transfer cost is assumed to be proportional to the size and number of the data elements transferred. Thus, the cost of switching between machines during the execution of a program is not a constant cost, but depends on data transfers needed as a result of the switch.

Because the mode-switching costs are non-constant, determining a minimum-cost assignment of machines to program segments is not straightforward. The cost of executing a given program segment depends on previous machine selections and associated data transfers made in the past. Exhaustively testing each possible mapping of program segments to machines provides a minimum-cost schedule, but because the number of possible mappings grows exponentially with the number of segments, this approach is not computationally feasible.

The BBMS approach, which provides the optimal sequence of modes under the constant switching cost assumption, is used here as a basis for a heuristic method for associating parallel machines with data-parallel program segments in an environment with non-constant switching costs. The heuristic provides an efficient approach for selecting near-optimal mappings of program segments to machines. Simulation results based on randomly generated parallel program behaviors indicate that good assignments are generally provided by the heuristic.

The rest of the paper is organized as follows. Underlying assumptions and basic terms are defined in Section 2. Section 3 overviews the BBMS framework and isolates the specific area of investigation. Also included in Section 3 is an explanation of why the BBMS must be extended for the case of data-location (i.e., non-constant) machine-switching costs. In Section 4, a heterogeneous program execution model is introduced, a static heuristic developed, and a series of simulations is presented to validate the approach.

2. Machine and Language Model

In a heterogeneous system [8], different types of parallel machines can be used to perform a single task. One

This research was supported by the Office of Naval Research under grant number N00014-90-J-1937, by Rome Laboratory under contract numbers F30602-92-C-0150 and F30602-94-C-0022, and by the National Science Foundation under grant number CCR-9202807.

0-8186-5592-5/94 $3.00 © 1994 IEEE

example of a heterogeneous environment is a mixed-mode machine [14], a single machine which is capable of operating in either the SIMD or MIMD mode of parallelism and can dynamically switch between modes at instruction-level granularity with relatively small overhead, e.g. PASM [3], Triton [13], and OPSILA [4] (limited to SIMD/SPMD). Another type of heterogeneous system is a suite of independent machines of different types interconnected by a high-speed network, referred to as a mixed-machine system. Unlike mixed-mode systems, switching execution among machines in a mixed-machine system requires measurable overhead because data may need to be transferred among machines.

Mixed-machine systems considered here are assumed to have high-speed connections among machines that make decomposition at the sub-program level feasible. The emphasis for this study is on machines interconnected using a current or future high-speed network. Whenever the term "SPMD machine" is used, recall that any MIMD machine can operate as an SPMD machine.

Each system in a mixed-machine suite is assumed to have a physically distributed memory. Thus, each processor in the system is paired with a memory, forming a processing element (PE).

Applications for this study are assumed to be data parallel [9], i.e., the data for a program is distributed among the PEs, and the algorithm exhibits a high degree of uniformity across the data [10]. It is assumed for mixed-mode machines that all PEs are in the same mode at any point in time (i.e., SIMD versus SPMD). Similarly for mixed-machine systems, a job is actively executed on only one machine at a time (i.e., only one SIMD or SPMD machine is being used for the program at any one time). It is assumed for this study that all machines in the system are available for the execution of any program segment. All PEs in a given machine must be synchronized at program segment boundaries before an inter-machine transfer can occur. Although the focus here is on SIMD versus SPMD machines, the approach can also be used to select among two or more machines of the same class (e.g., among several different SIMD machines).

One way of programming machines that are capable of mixed-mode and mixed-machine parallelism is by using a mode-independent language, i.e., a language whose syntactic elements have interpretations under more than one mode of parallelism. An example of a mode-independent language is the Explicit Language for Parallelism (ELP), currently under development at Purdue University [12]. The ELP syntax is based on C, and allows the programmer to specify SIMD, MIMD, SPMD, and mixed-mode operation within a program. A goal of ELP is to provide uniformity with respect to the SIMD and SPMD modes of parallelism by having interpretations for all the elements of the syntax within both of these modes that are identical in semantics. This is an important characteristic because it allows a data-parallel algorithm to be coded in a mode-independent manner, producing a data-parallel program for which: (1) only SIMD code is generated, (2) only SPMD code is generated, or (3) execution mode specifiers can be added to facilitate mixed-mode experimentation. Other related unifying parallel language studies include [5], [13], and [16].

3. The BBMS Framework

3.1. Framework Overview

To determine the best machine for each portion of a parallel algorithm, the program is first transformed into a flow–analysis tree, which is a tree whose structure represents the scope levels within the algorithm. The flow-analysis tree is then used to create a trade-off tree that has a structure identical to the flow-analysis tree, but indicates the machine in a mixed-machine system on which to execute each portion of the program. For both trees, the root node represents the scope of the entire program, the non-leaf nodes correspond to control constructs and data-conditional constructs, and the leaf nodes of the tree represent parallel or scalar code blocks.

Code blocks are the parallel equivalent of the serial basic blocks described in [1]. Blocks of code are identified by their leading statements, called leaders. The first statement in a program is a leader, any statement that becomes the target of a conditional or unconditional branch at the machine code level is a leader, and any statement that follows a conditional branch at the machine code level is a leader. In the approach described here, in addition any statement requiring synchronization and any statement that follows a statement requiring a synchronization is a leader, and any statement requiring an inter-PE data transfer and any statement that follows an inter-PE data transfer is a leader. This is an important distinction from the serial definition of a basic block, because synchronization points within an algorithm indicate when PEs are idle, waiting for one or more other PEs to arrive at the synchronization point.

Initially, only information about the execution times of the leaf nodes is assumed to be known. The technique proposed in [15] combines this known information to arrive at comparative mixed-mode execution times for the intermediate nodes, using a limited set of language constructs. Intermediate nodes, which correspond to looping and data-conditional constructs, are implemented such that they begin and terminate execution in either SIMD or SPMD, but may switch modes zero or more times during execution. For non-leaf nodes corresponding to descendents of data-conditional constructs, mode changes are disallowed, because these changes can translate into operations that cannot be implemented without excessive execution time overhead [15]. For a sequence of nodes that are children of a looping construct, the modes of the nodes can differ. However, the first and last nodes must be implemented in the

same mode of parallelism or a mode switch is added before the first node. Thus, the only subtree that does not necessarily begin and terminate in the same mode is the node corresponding to the root of the entire program.

It is assumed that both the SIMD and SPMD execution times associated with each leaf node are known constants. The case where execution times of the operations are assumed to be random quantities with known probability distributions (which impacts the distribution of execution time of combined adjacent blocks) is a consideration that will be examined in future studies. In [15] and in this paper, iterative loop bounds and information regarding the probability of data-conditional outcomes is assumed known or estimated statistically.

Figure 1 illustrates how a program is transformed into a preliminary flow-analysis tree, and then used to generate a trade-off tree (details are in [15]). At each level in the tree, the sibling blocks are represented in the same order (left to right) as they appear in the program. In the final trade-off tree, only machine selection information is retained.

In mixed-mode and mixed-machine systems, it is possible for some portions of a single parallel program to be implemented in one mode or machine, and other portions of the same program to be executed in a different mode or machine. This is referred to as a mixed-algorithm. Leaf blocks are assumed to be implemented completely in one mode/machine. Given the definition of a block, it is assumed that there is no benefit in changing modes except at block boundaries.

3.2. With Constant Switching Costs

Consider the mixed-mode case, where a node may be executed in either SIMD or SPMD mode, and there is a cost (assumed for the moment to be constant) associated with switching from one mode of parallelism to the other. One portion of the block-based framework developed for this case is the approach taken to reduce a set of children in the flow-analysis tree under a common parent so that it is represented by a single node at the location of the parent node in the original tree. An ordered pair (T_i^{SIMD}, T_i^{SPMD}) is assigned to each leaf block, representing the execution time required to perform block i in SIMD or SPMD mode, respectively. Leaf nodes with a common parent are executed in order from leftmost node to rightmost node. For each non-leaf node n within the flow-analysis tree, (T_n^{SIMD}, T_n^{SPMD}) corresponds to the time required to perform the entire subtree rooted at n beginning and ending in either SIMD or SPMD mode, with no restriction on the number of mode changes that occur within the subtree.

Because there are two choices of mode for each of m children of a parent node, there are 2^m possible ways to execute the sequence of children nodes. Exhaustively testing all possible implementations to find the minimum mixed-algorithm execution time for a node with m children would therefore require testing 2^m combinations. For nodes with many children, this approach is impractical. A non-exponential method of finding the minimum execution time is needed when m is large.

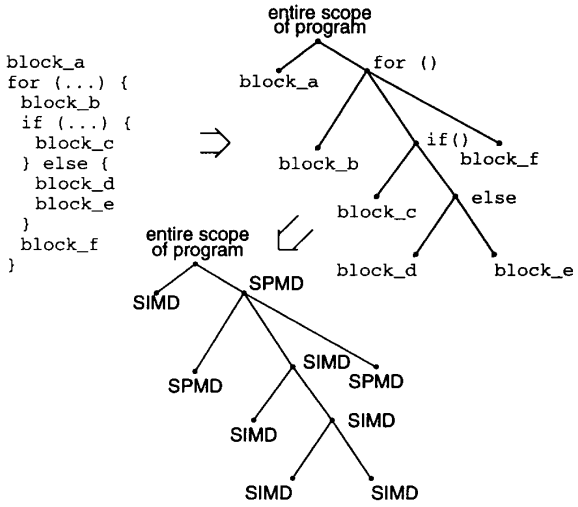

Figure 1: Example of transformation from program to flow-analysis tree to trade-off tree.

One efficient way of computing (T_n^{SIMD}, T_n^{SPMD}) is to transform the m children of node n into a multistage optimization problem. In multistage optimization problems, proceeding to stage $j+1$ is possible only by passing through stage j. There is a cost associated with proceeding from stage j to stage $j+1$ dependent on the initial state (at stage j) and the final state (at stage $j+1$). In a multistage optimization graph, each state in each stage is represented by a vertex, and edges connecting vertices in stage j to vertices in stage $j+1$ indicate valid state transitions, each with an associated cost. The goal of a multistage optimization problem is to find the minimum cost path between the initial and final stages, e.g., see [2, 6].

Multistage optimization problems can be solved using Moore's algorithm [11] (a variant of Dijkstra's algorithm [7]) in $s-2$ iterations for an s-stage problem. In each iteration, two successive stages of the problem are reduced to a single stage, so that at the end of $s-2$ iterations, only the initial and final stages remain, with connecting edges indicating the minimum cost from each of the states in the initial stage to each of the states in the final stage. By redrawing the set of m children nodes of a parent as a multistage optimization problem, the minimum cost mapping of parallel modes to m leaves can be found for the corresponding ordered pairs and mode switching costs in $O(m)$ time.

An example of a general multistage optimization problem and its solution is illustrated in Figure 2. Initially, there are four stages, numbered from 0 to 3. In the first iteration, for each vertex in stage 0, a minimum path is found to each vertex in stage 2 (passing through stage 1). Stage 1 is then removed from the graph, and new edges are drawn from

vertices in stage 0 to vertices in stage 2, indicating the minimum cost to proceed from stage 0 to stage 2 for each possible source/destination pair. This reduction procedure is applied again, and the problem is reduced to two stages, indicating the minimum cost to proceed from each initial state to each final state. By recording the minimum paths selected during each iteration of the algorithm, the path through the entire multistage graph resulting in the minimum cost for each initial/final pair is obtained.

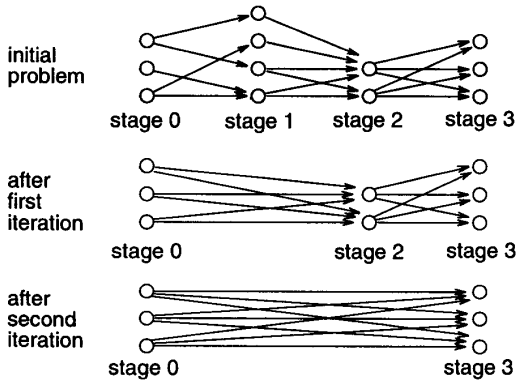

Figure 2: Example of a general multistage optimization problem and its solution.

To find the minimum execution time for all the children of a node in the SIMD/SPMD trade-off tree, let a stage and the edges exiting that stage in a multistage optimization graph correspond a single child node. Let each mode of parallelism represent a separate state within each stage. The cost associated with each edge corresponds to the cost of performing that node in SIMD mode or SPMD mode. Then, before the representation of each child node in the multistage graph, include a stage and associated edges representing the cost of a possible mode switch between nodes. If the mode switching cost is assumed to be constant ($\underline{C^{SIMD}}$ and $\underline{C^{SPMD}}$), then all the arc weights in the multistage graph are independent of past decisions, and the minimum cost path can be determined.

As an example of the transformation of a flow-analysis subtree into into a multistage optimization problem, consider the transformation illustrated in Figure 3(a). To find the minimum mixed-algorithm execution time for a sequence of sibling nodes, each node is modeled as three stages of a multistage graph, numbered 0 to 2 in Figure 3(a), and then joined together to form a multistage optimization graph for the entire sequence of nodes. In each stage, the upper vertex represents SIMD mode, and the lower vertex represents SPMD mode. For stage 0 and its associated edges, a possible mode change is represented. The edge from the upper vertex in the stage 0 to the lower vertex in stage 1 is labeled with the cost of switching from SIMD to SPMD mode. Similarly, the cost of switching from SPMD to SIMD is indicated as the label for the edge from the lower vertex in stage 0 to the upper vertex in stage 1. There is zero cost for remaining in the same mode of parallelism. The edges from stage 1 to stage 2 in Figure 3(a) represent the cost of executing the node in each mode (or for a loop construct starting and stopping in that mode). Figure 3(b) illustrates the graph for a SIMD/SPMD trade-off tree with three children. The ordered pair $(T^{SIMD}_{mixed\text{-}4}, T^{SPMD}_{mixed\text{-}4})$, which represents the optimum mixed-mode implementation of the subtree, is obtained from the solution of the multistage optimization problem.

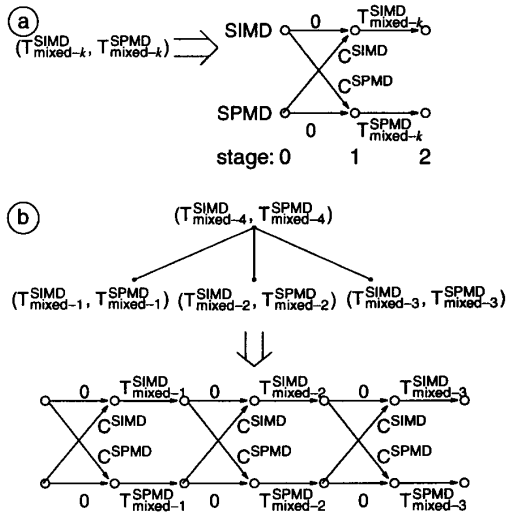

Figure 3: Transformation from flow-analysis tree to multistage graph.

A node with m children is represented by a multistage optimization problem with $2m+1$ stages: m stages to represent the nodes, m stages to represent possible mode switches, and one stage to represent the final states at the end of the multistage optimization graph. Thus, $2m+1-2 \cong 2m$ iterations of Moore's algorithm are required to determine a solution. For each iteration corresponding to a possible mode switch, $2^2 = 4$ comparisons and $2^2 = 4$ assignments are needed, while for each iteration corresponding to a node execution, no comparisons and $2^2 = 4$ assignments are needed. Thus, finding the minimum mixed-algorithm execution time by this approach has a sequential time complexity of $8m + 4m = 12m = O(m)$ time.

By recording the minimum paths selected at each iteration, the mode of parallelism for each stage can be determined. Each edge in the minimum-cost path selected corresponds to performing a node in SIMD, performing a node in SPMD, or performing a mode switch. When the multistage optimization is completed for all the children of a node in the SIMD/SPMD trade-off tree (under the constant mode-switching cost assumption), the execution time information is used to generate the execution time for the parent node. The multistage approach is then applied to the parent node and its siblings. In this way, the analysis works

upwards through the trade-off tree until execution costs are obtained for the root node.

3.3. With Non-Constant Switching Costs

In mixed-machine heterogeneous systems, the cost to switch from one machine to another is assumed to depend primarily on the cost of moving the required data between machines. A given machine may contain a data set because it was initially loaded there, it was received from another machine, or it was generated by that machine. (It is assumed that the assignment of program code will be determined, and code distributed, prior to execution time.) Thus, unlike the assumption of the previous subsection, the cost of switching execution from one machine to another is dependent on which machine(s) contain the data structures that are required to execute the next block, which in turn is dependent on the machine choices made for previous blocks. Alternatively, the effectiveness of a current machine selection and associated data movement is impacted by the data movement required for the optimal machine assignment for a later program segment. Under these conditions, it can be shown that the multistage optimization approach described in the previous section does not directly apply because the cost of switching machines is not a known constant. However, as to be described, it forms the basis of a useful heuristic in this situation.

4. Extension of BBMS

4.1. A Simplified Parallel Program Behavior Model

Parallel programs are divided into a sequence of individual blocks (B_0, B_1, B_2, ...). Each block is considered exactly once, in a sequence beginning with the first block and ending with the last. In this way, the execution of a sequence of sibling nodes under a common parent node is modeled. A block sequence itself may, as the flow-analysis tree is reduced, form an interior node as part of another construct, but in this section the focus is on determining a low-cost path only for the immediate block sequence under a single parent node in a mixed-mode heterogeneous system.

Each block B_i in the parallel program can be executed on one of several machines. For each possible machine, there is an associated execution time assumed to be known a priori. Mixed-machine systems consisting of two parallel machines are considered here, but the results can be generalized to include systems with more than two machines.

Associated with a parallel program is a set of data structures that are used by the program during execution. Each data structure j is assigned a weight C^j, indicating the cost of moving that data structure to another machine in the system. Although the time cost of moving a data structure can be related to the source and destination machines, to clarify the concepts presented, the time cost for each data structure is assumed to be independent of the source and destination of the transfer. The analysis can be extended to include these source/destination dependent costs.

Also associated with each data structure is a location attribute. The location of a data structure can be enumerated as being on no machine, on a single machine, or on a set of machines. A data structure having no location corresponds to a structure that may be available at an I/O device for the system, but not available on any machine. According to the machine selection policy, the data structure can be assigned to the portion of the heterogeneous environment where it is first used. A structure that has multiple locations is defined to have the same state at each location. This may correspond to a data set that is only read by a block.

A data location (DL) table is defined as a table with entries for each data structure used within a parallel program. The DL table initially corresponds to the starting location for each of the data structures used. As the static source-code based analysis of the program proceeds, the DL table is updated to reflect the movement of data structures that would occur as a result of machine choices made thus far in the program.

To differentiate the state of the DL table for different portions of the program, a subscript is added to indicate the block that corresponds to the state of the table before execution of that block. For example, DL_0 corresponds to the initial location of data structures in the heterogeneous system before B_0 is executed, and DL_1 reflects the changes in the location of data structures required as a result of B_0.

Associated with each block in the program is a data use (DU) table listing each data structure used in that block, as well as a usage type, which designates the way in which that structure is modified by the block associated with the DU table. Possible usage types include the read type, where a structure is used but not altered by a block; the create type, where the values of a structure are first generated; and the modify type, where one or more elements of the structure are modified, based on on the current state of the data structure. Other usage types are also possible.

As an example, consider the representation of a parallel program given in Figure 4. The program consists of three blocks, labeled B_0, B_1, and B_2, illustrated vertically in the center of the figure. The blocks labeled "begin" and "end" in the figure do not correspond to executable code, but instead to initial and final states. B_0 requires a time cost of 20 to execute on machine X, and 2 on machine Y. B_1 and B_2 each require 10 to execute on machine X, and 10 on machine Y. Each DU table is shown to the left of its corresponding block. In B_0 the data structure P is modified, in B_1 the data structure Q is created, and in B_2 the data structures P and R are modified.

The DL tables are shown for each step during the planned parallel execution. Before execution there are three data structures, P, Q, and R, initially placed on machine X, as indicated by DL_0. Assume the first block is be executed on machine Y. It modifies data structure P and this change is reflected in DL_1. Assume B_1 is also to be executed on

machine Y. It creates data structure Q. Therefore, DL_2 indicates that Q is now also on machine Y. It is assumed B_2 is executed on machine X. Because it modifies P, P must be moved from Y to X, as indicated in DL_2 and DL_{final}. Block B_2 also modifies R, but R is already in X (see DL_2) so no data transfer is needed.

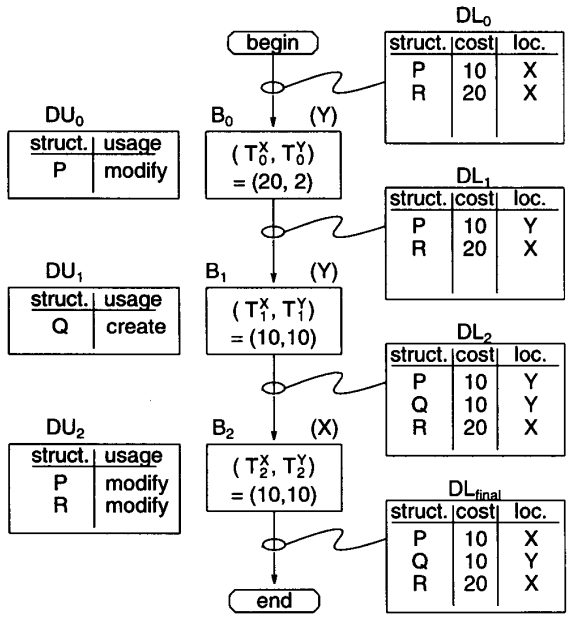

Figure 4: Simplified model of parallel program behavior.

4.2. Heuristic Based on BBMS

Given a sequence of blocks (B_0, B_1, B_2, ...), a set of data usage tables and associated costs to transfer the data structures between machines in a heterogeneous system, the goal to to find an assignment of machines to blocks that results in the minimum execution time. As discussed in Section 3, the multistage optimization technique presented in [15] finds the optimum mapping of modes to program segments when the cost to switch modes is constant, but the technique cannot be applied directly to the mixed-machine case where non-constant data transfer costs are considered.

To reduce the complexity of finding an assignment of machines to program segments, a heuristic approach is used. In this approach, four independent sets of DL tables are used to track possible routes through the multistage optimization graph. To find the shortest path through a completely-weighted multistage graph with s stages, $s-2$ iterations are required. During each iteration, the shortest path from each state in the initial stage to each state in the next stage is determined. Because there are only two initial states (representing machine X and Y) and two states in the next stage, only four paths are considered at each iteration. In the heuristic extension for non-constant switching costs, a DL table is associated with each path in the solved portion of the multistage graph (Figure 5). Each DL table represents the location of all data structures needed by the parallel program that results from choosing the path designated by the corresponding arc in the solved portion of the multistage graph. Additionally, because programs with large data sets often are found to execute in minimum time without switching machines, the single-machine implementation is always considered for each machine in the heterogeneous system.

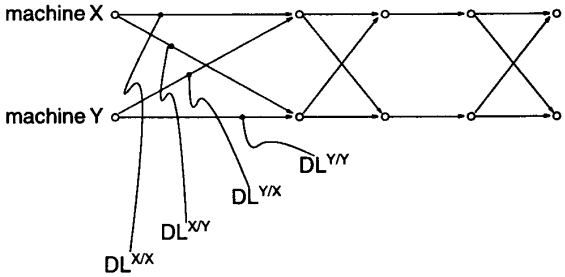

Figure 5: Heuristic building on the multistage technique.

4.3. Simulation Study

Examples exist for which the heuristic selects machine mappings that are not optimal. This is because machine choices, and hence data transfer decisions, do not consider the data location needs of blocks that follow. To determine if the heuristic generally provides good results, a simulation study has been performed. In the study, randomly-generated parallel program behavior representations were used to test the proposed heuristic. Then, the resulting assignment of machines was compared to the optimum assignment (found by exhaustive search) to determine the validity of the machine-selection decision. Parameters of the parallel program behavior being modeled are then modified to determine the effect of the parameters on the quality of the assignment decision.

One of the goals of the simulation study is to explore how the heuristic performs in a general-purpose environment. To examine the behavior of the approach for a variety of program structures, and because it is unclear what is meant by a "typical" parallel program, randomly-generated parallel program representations are used instead of specific program traces.

In the simulation, parallel program behaviors are generated using the simplified model in Subsection 4.1. No parallel code is generated; only parameters needed by the model. For each leaf block B_i in the simulated program, an ordered pair (T_i^X, T_i^Y) is randomly assigned with uniform distribution over a specified range. No correlation between T_i^X and T_i^Y is assumed.

The number of data structures used by the program is then assigned, also with uniform distribution over a specified range. Each data structure is given an initial loca-

tion and weight (cost to transfer), again uniformly over a specified range without correlation. A DU table is then assigned to each block in the program, with data structure requirements and usage types also assigned. Again, all the values are uncorrelated. This is important because of interest is how the heuristic performs in the absence of predictable events, and because the characterization of "typical" parallel programs remains unclear in the general purpose computing field.

After the parallel program behavior is generated, four separate analyses are performed with the following policies: best, worst, heuristic, and random, each providing an assignment of machines to each block in the program. The best policy determines the optimum (minimum time cost) assignment of machines to program blocks by exhaustively testing every possible mapping. Because this approach is so expensive, simulations can be performed for only a limited number of stages (twelve in this study). The worst policy finds the most costly mapping, again by exhaustive search, and is used as a lower bound. The heuristic policy implements the policy described in Subsection 4.2. The random policy makes machine assignments by using the bit-wise representation of a randomly-generated integer to serve as a map for making machine selections.

Three sets of simulations are included here, differing in the ratio of transfer time to computation time. In each set, the number of stages in the generated parallel program behaviors is controlled, and is varied from three to twelve. At each setting of the number of stages to be simulated, 2^8 simulated parallel program behaviors were generated. For each generated program, each of the four policies were used to determine an assignment of machines to blocks. Then, the time cost for each selected mapping was determined.

Because the actual time cost for each parallel program varies significantly, the information for each heuristically and randomly selected mapping was normalized against the best and worst mapping for that algorithm. Specifically, the heuristic norm for a parallel program k, termed h-norm, is determined by the following definition:

$$\text{h-norm}_k = 1 - \left[\frac{(\text{heuristic}_k - \text{best}_k)}{(\text{worst}_k - \text{best}_k)} \right]$$

The random norm (r-norm) is defined similarly.

Using this definition, an h-norm of 1 indicates that the optimum mapping was selected, while an h-norm of 0 indicates that the worst possible mapping was selected. The results for all generated programs for each parameter setting are then averaged.

In each of the three simulation sets, the relative size of data structures (i.e., data transfer cost) is controlled in relation to the computational cost of each stage. This is accomplished by restricting the range over which data structure sizes and execution times are chosen. In the first set of simulations, the data/execution cost ratio is set at 1/10, indicating that data size is generally small in relation to the execution

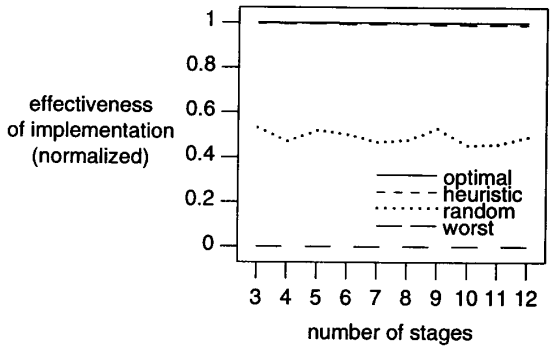

Figure 6: Effectiveness of machine selection policies for data/execution ratio of 0.1.

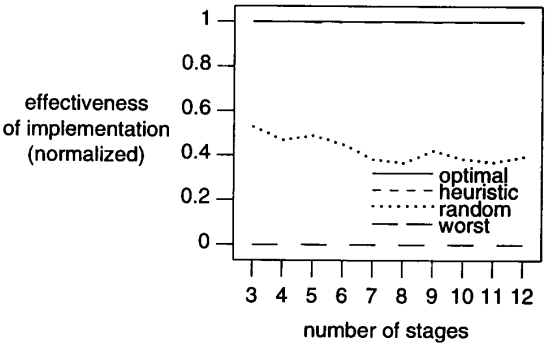

Figure 7: Effectiveness of machine selection policies for data/execution ratio of 10.

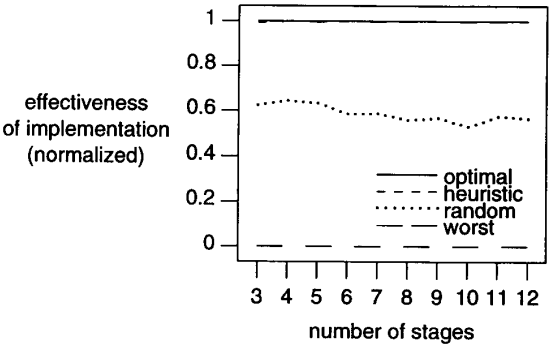

Figure 8: Effectiveness of machine selection policies for data/execution ratio of 1000.

requirements for the algorithm. For this set of simulations the heuristic is expected to perform well, because the effect of transferring data between machines is not significant, and therefore results should be similar to the constant-cost case.

Results for this simulation are shown in Figure 6. The horizontal axis in the graph indicates the number of stages in the simulated parallel program behavior. The vertical axis indicates the relative worth of the machine-selection decisions for the simulated program behaviors. The solid line in

the graph represent the "best" policy, which is a constant value of 1. Similarly, the "worst" policy (large-dashed line) is a constant value of 0. The dotted line indicates the relative worth of the "random" policy. The heuristic policy is represented by the small-dashed line and performs close to the optimal. For many of the simulated cases, the heuristic policy chooses the same mapping as the "best" policy. Each data point in the graph represents the average of the r-norms and h-norms of 2^8 simulations.

In the second set (Figure 7), the data/execution ratio is set to 10. Again, the heuristic performs very well.

In the third set of simulations, shown in Figure 8, the data/execution ratio is 1000. Parallel programs for these parameter settings tend to become polarized in that they are normally implemented entirely in one machine, because switching from one machine to another normally incurs a very high data transfer penalty. Even under these extreme conditions, the machine selection heuristic still selects good mappings.

In Table 1, the percentage of optimal execution time required for both the random assignment policy and the heuristic assignment policy for the 12-stage simulation is given. As the data/execution cost ratio increases, the random assignment policy results in significantly higher cost assignments, while the heuristic policy finds mappings that are close to optimal (100%).

data/execution cost ratio	random assignment	heuristic assignment
0.1	140.7%	100.6%
10	449.1%	100.1%
1000	515.6%	100.5%

Table 1: Percentage of optimal execution time for random and heuristic assignments (# stages = 12).

5. Summary

The problem of minimizing the execution time of programs within a mixed-machine heterogeneous environment has been considered. A heuristic for determining the machine to execute each portion of a parallel program is examined, with an emphasis on efficiently determining the best assignment of machines to program segments in the presence of data-location dependent machine switching costs. Results of simulated parallel program behaviors indicate that good assignments are possible without resorting to exhaustive search techniques.

References

[1] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1986.

[2] J. K. Antonio, W. K. Tsai, and G. M. Huang, "A highly parallel algorithm for multistage optimization problems and shortest path problems," *J. Parallel and Distributed Computing,* Vol. 12, No. 3, July 1991, pp. 213-222.

[3] J. B. Armstrong, D. W. Watson, and H. J. Siegel, "Software issues for the PASM parallel processing system," in *Software for Parallel Computation,* J. Kowalik and L. Grandinetti, eds., Springer-Verlag, Berlin, Germany, 1993, pp. 134-148.

[4] M. Auguin and F. Boeri, "The OPSILA computer," in *Parallel Languages and Architectures,* M. Consard, ed., Elsevier Science, Holland, 1986, pp. 143-153.

[5] J. C. Browne, M. Azam, and S. Sobec, "CODE: A unified approach to parallel programming," *IEEE Software,* Vol. 6, No. 4, July 1989, pp. 10-18.

[6] A. E. Bryson and Y.-C. Ho, *Applied Optimal Control,* Hempshire, Washington, DC, 1975.

[7] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mithematik,* Vol. 1, 1959, pp. 269-271.

[8] R. F. Freund, "SuperC or distributed heterogeneous HPC," *Computing Systems in Engineering,* Vol. 2, No. 4, 1991, pp. 349-355.

[9] W. D. Hillis and G. L. Steele, Jr. "Data parallel algorithms," *Communications of the ACM,* Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.

[10] L. H. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of Parallel Algorithms,* L. Jamieson, D. Gannon, and R. Douglass, eds., MIT Press, Cambridge, MA, 1987, pp. 65-100.

[11] E. F. Moore, "The shortest paths through a maze," *Int'l Symp. Theory of Switching,* 1957, pp. 285-292.

[12] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler," *IEEE Trans. Parallel and Distributed Systems,* Vol. 4, No. 2, Feb. 1993, pp. 222-234.

[13] M. Philippsen, T. Warschko, W. F. Tichy, and C. Herter, "Project Triton: towards improved programmability of parallel machines," *26th Hawaii Int'l Conf. System Sciences,* Vol. 1, Jan. 1993, pp. 192-201.

[14] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems," *IEEE Computer,* Vol. 25, No. 2, Feb. 1992, pp. 54-63.

[15] D. W. Watson, H. J. Siegel, J. K. Antonio, M. A. Nichols, and M. J. Atallah, "A framework for compile-time selection of parallel modes in an SIMD/SPMD heterogeneous environment," *Workshop on Heterogeneous Processing (WHP '93),* Apr. 1993, pp. 57-64.

[16] H. P. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: a tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing,* Vol. 6, No. 1, Jan. 1988, pp. 1-18.