A Framework for Compile-Time Selection of Parallel Modes in an SIMD/SPMD Heterogeneous Environment

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

†Parallel Processing Laboratory, School of Electrical Engineering ††Computer Sciences Department
Purdue University, West Lafayette, IN 47907 USA

Abstract

A 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 is presented. Given a data-parallel program in a language whose syntax is mode-independent, and empirical information about instruction execution time characteristics, the long-term goal is to determine at compile time an implementation that results in an optimal execution time for a heterogeneous system capable of SIMD and SPMD parallelism.

1: Introduction

In general, writing effective programs for parallel computers remains a complex and difficult endeavor. In many cases, algorithms that work well under a serial execution model require reformulation for implementation on a parallel system. The problem is compounded by the rich diversity of parallel architectures available, each with its own advantages and disadvantages. Subsequently, parallel languages for these systems vary substantially, as vendors are (justifiably) concerned with maximizing performance for their products. To address this diversity in parallel programming environments, there is a need for the development of unifying parallel machine models [24]. One of the challenges of compilers and compiler-related tools is, given a machine-independent parallel language, to generate executable code for a variety of parallel architectures, and to identify those specific parallel architectures for which the program is well-suited.

One portion of this problem is developing a method for estimating the relative execution time of a data-parallel algorithm in an environment capable of the SIMD and SPMD modes of computation. The SPMD (Single Program - Multiple Data) mode of parallelism is a restriction of the MIMD mode where each processor is executing the same program asynchronously with respect to the other processors (and follows its own control path through the code). Given a data-parallel program written in a language whose syntax is mode-independent, and given empirical information about instruction execution characteristics, the goal is to make a compile-time determination of the implementation that results in an optimal execution time on a heterogeneous system capable of both the SIMD and SPMD modes of parallelism. Heterogeneity within a single machine and among a suite of machines is considered. This paper is a summary of [29].

One aspect of current research in parallel optimization and analysis techniques in parallel and heterogeneous processing is directed towards improving execution time in a parallel environment by performing compile-time analysis of parallel programs [e.g., 11, 14, 17, 23, 28], while another aspect of current research is the study of heterogeneous systems, which exploit the diverse computational requirements of traditional supercomputing problems by the selection and use of different types of machines for the computation required [15]. Several studies [e.g., 4, 5, 13, 26] have examined the implementation of parallel programs for heterogeneous and mixed-mode systems that can operate in either the SIMD or MIMD mode of parallelism. A common result from these studies is that cases exist where the execution time of a parallel application can be reduced by exploiting the mode of parallelism that best matches each portion of the program. A third aspect of recent academic and commercial interest in parallel computing systems is in the development of new programming languages and refined models of programming [e.g., 8, 10, 21, 30]. The relationship of this study to other work in the area, as well as a detailed development of background material, is contained in [29].

Mode-independent languages make it possible for the compiler to select the appropriate parallel mode. This study proposes a framework for the compile-time analysis of execution time for data-parallel algorithms. These algorithms are assumed to be written in a mode-independent language, and to be implemented in an SIMD/SPMD heterogeneous environment. The proposed framework draws on the above-mentioned previous work in the areas of compile-time analysis, heterogeneity, and programming models.

To illustrate underlying concepts for the methods presented, the model is first developed for the case where a parallel program is to be implemented in either pure SIMD mode or pure SPMD mode (distributed memory machines are assumed). The more general case of a single program that employs both modes is then considered. For ease of presentation, the programming model presented here is restricted to basic processor operations and elementary control-flow constructs. Statistical information about individual operation execution times and paths of execution through a parallel program is assumed. This basic model can be enhanced to include other language constructs.

The techniques presented here represent a framework for studying this complex and difficult problem. A secondary

This research was supported by the Office of Naval Research under grant number N00014-90-J-1937, and by Rome Laboratory under contract number F30602-92-C-0150.

††† The author is currently with NCR Corp., San Diego, CA.

goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments.

2: Machine and language model

In a heterogeneous system [15], different types of parallel machines can be used to perform a single task. One example is a mixed-mode system, in which the processors are 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 [25], Triton[22], and OPSILA [3] (limited to SIMD/SPMD). Another type of heterogeneous system is a suite of independent systems with machines of different types interconnected by a high-speed network. This will be referred to as a mixed-machine system.

The mixed-mode machine model assumes a physically distributed memory. Similarly, each system in a mixed-machine suite is assumed to have a physically distributed memory. Thus, each processor is paired with a memory, forming a processing element (PE).

It is assumed for mixed-mode machines that all PEs are in the same mode at any point in time (i.e., SIMD versus MIMD). Similarly, for this study it is assumed that for mixed-machine systems, a job is being actively executed in only one machine at a time (i.e., only one SIMD or MIMD machine is being used for the program at any one time).

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), under development at Purdue University [21]. The ELP syntax is based on C, and allows programmers to specify the SIMD, MIMD, and SPMD modes of parallelism within a program.

The trade-offs between the SIMD and MIMD modes are discussed in [6, 13, 25]. For example, conditional statements in a program can degrade an SIMD implementation, because the CU (control unit) must broadcast the "then" and "else" clauses of conditional statements one after the other. However, the cost to synchronize PEs in MIMD mode is greater than in SIMD (where synchronization is implicit). While the details of the inter-PE transfer protocol in both SIMD and MIMD modes are implementation dependent, there is generally less overhead associated with SIMD, because of the implicit PE synchronization.

Although it is often clear in which mode a sequence of instructions should be implemented, this is not the case when counteracting trade-offs are involved. For example, a data-conditional clause may contain instructions that perform network transfers. Studies examining the implementation of algorithms on mixed-mode systems have been performed [e.g., 4, 5, 7, 13, 16]. One long-term goal of these efforts has been to increase the understanding needed to develop automatic, compile-time mapping of parallel modes to algorithm segments. In Sections 3 and 4, a methodology for analyzing data-parallel programs to determine how to perform this mapping is examined.

3: Single-mode selection

3.1: Assumptions and definitions

A methodology is presented here to estimate the execution time of a data-parallel algorithm, written in a mode-independent language, that can be implemented in either purely the SIMD or purely the SPMD mode of parallelism. A technique for determining whether the program should be executed in SIMD or SPMD by quantifying the trade-offs in Section 2 is proposed. (The impact of other trade-offs is discussed in [29].) This framework can be used to select either an SIMD machine or an SPMD machine (which may be a more general MIMD machine) in a mixed-machine system, or to select the optimal single-mode mapping on a mixed-mode computer. Many of the concepts developed in this preliminary section are needed for the analysis in Section 4, where the use of both SIMD and SPMD within the same program is considered.

Applications for this study are assumed to be data parallel [18], i.e., the data for a program is distributed among the PEs, and the algorithm exhibits a high degree of uniformity across the data [19]. This is in contrast to function (or control) parallelism [27], where each PE executes a unique program.

Syntactic elements of the mode-independent language in which the algorithm is coded will be referred to as operations. Operations in a language represent the most explicit level at which program representation is identical for each mode of parallelism.

The execution time of an operation may be fixed (and known) or data dependent. For the case of data-dependent operations, a probabilistic model is introduced in [29] to estimate the expected execution time for a block of operations. It is assumed that the necessary statistical information can be estimated empirically and is known a priori.

When comparing machines in a mixed-machine system, execution times of operations can differ significantly. Even in a mixed-mode machine, execution times of the same operation, e.g., an inter-PE data transfer, under SIMD and SPMD modes may differ.

To estimate the overall execution time of a parallel program, code is examined in terms of constructs and blocks. Constructs include control constructs, such as looping structures and function calls, and data-conditional constructs, such as `if-then-else` and `case` statements. For the analysis here, the set of permissible control constructs is restricted to `for` loops for which the number of iterations, Q, is assumed to be known at compile time. Furthermore, because programs under consideration exhibit characteristics that make them candidates for implementation in SIMD mode (in which all PEs iterate the same number of times) it is also assumed that all PEs executing a loop in SPMD mode iterate the same number of times. In a practical sense, an expected value for Q may be obtained from the program source, either directly as a constant in the program, or as

information provided to the compiler by the programmer in the form of a compiler directive. Alternatively, a sufficiently accurate expected value for Q might be obtained empirically by executing a prototype version of the parallel program on sample data sets. While time-consuming, this empirical approach may be reasonable for production codes.

The set of data-conditional constructs is limited here to `if-then-else` constructs, where all PEs may or may not perform the same blocks of code, depending on the outcome of a conditional test of local PE data. As in the case for operation execution times and loop bounds, it is assumed that the necessary statistical information regarding the probability that a branch will be taken can be estimated empirically or is obtainable by compiler directives provided by the programmer.

Future work will examine relaxing the restrictions to include other constructs, e.g., `while` and `case` statements. It will also investigate methods for obtaining the statistical information assumed to be known here.

Code blocks are the parallel equivalent of 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, an additional constraint is used in the determination of blocks: 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 in [1], because synchronization points within an algorithm indicate when PEs are idle, waiting for one or more other PEs to arrive at the synchronization point.

Blocks consist of either pure scalar code or pure parallel code. Scalar blocks consist of instructions that, if performed in SIMD mode, would be executed on the CU. Parallel blocks consist of instructions to be performed on the PEs in SIMD mode.

3.2: Single-mode selection technique

To determine the best single mode for a given program, 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 an SIMD/SPMD trade-off tree. For both trees, the leaf nodes of the tree represent parallel and scalar code blocks, and the non-leaf nodes correspond to control constructs and data-conditional constructs. The root node represents the scope of the entire program. Initially, only information about the execution times of the leaf nodes is assumed to be known. The proposed technique involves combining this known information to arrive at SIMD and SPMD execution times for the intermediate nodes. The optimal mode determined for the root node represents the "best" mode for 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 is examined in [29] (this impacts the time of combined adjacent SPMD blocks).

Fig. 1 illustrates how a program is transformed into a preliminary flow-analysis tree. At each level in the tree, the sibling blocks are represented in the same order (left to right) as they appear in the program.

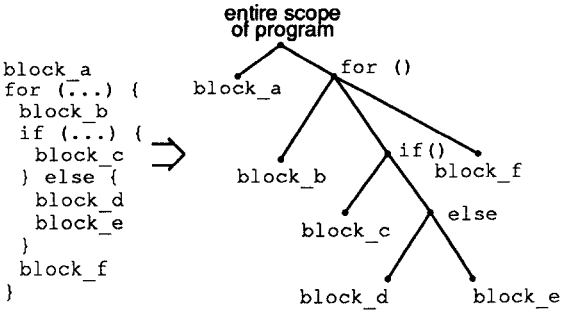

Fig. 1: Example preliminary flow-analysis tree.

To include the overhead required for `for` and `if` constructs, it is necessary to introduce additional nodes in the flow-analysis tree. The additional nodes for the previous example are illustrated in Fig. 2.

Associated with each `for` loop is a block required for initializing induction variables, and another block required to execute the induction step for the loop, perform an end-of-loop test, and conditionally branch back to the top of the loop. Because the initialization block is executed only once, it is represented as a separate child of the root node in Fig. 2, labeled `for_init`. Because the increment, test, and conditional branch are executed during each iteration of the loop, they are represented by a single block as the last child of the `for` node in Fig. 2, labeled `for_test`.

Associated with each `if` construct is a conditional test (`if_test` in Fig. 2), performed before either the `then` clause or the `else` clause is executed. There is also special processing required at the end of the `then` clause (`post_then` in Fig. 2). In SPMD mode, this is an unconditional branch performed on each of the PEs for which the condition is true. In SIMD mode, this corresponds to disabling those PEs for which the condition is true and enabling those PEs for which the condition is false (with nested conditionals handled appropriately). Similarly, at the end of the `else` clause in SIMD mode, those PEs for which the condition is true must be re-enabled (`post_else` in Fig. 2). Because there is no corresponding operation required in SPMD, the cost for executing the `post_else` in SPMD mode is 0. Because the `then` clause of the `if` construct is now composed of two blocks, an interior node is added representing the `then` clause, with leaves `block_c` and `post_then` as children.

After the final flow-analysis tree is formed, the following steps are performed to convert it to a SIMD/SPMD trade-off tree, as illustrated in Fig. 3:

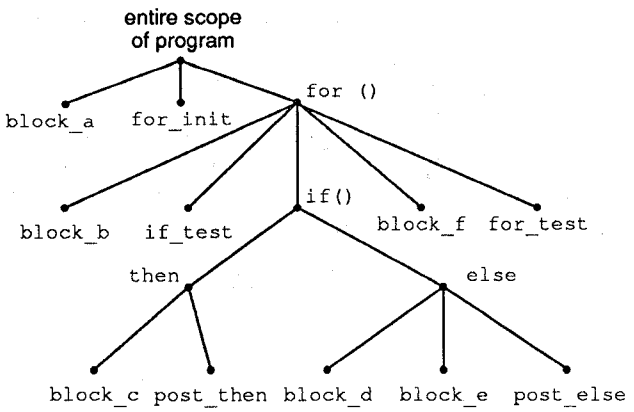

Fig. 2: Final flow-analysis tree modified to include overhead for `for` and `if` constructs.

1) For each leaf l of the tree, assign an ordered pair (T_l^{SIMD}, T_l^{SPMD}), which represents the times for executing the block associated with leaf l in SIMD mode and SPMD mode, respectively.

Steps 2 and 3 are then performed for each non-leaf node in the order of a depth-first traversal of the tree.

2) For each non-leaf node d corresponding to a *data*-conditional construct, assign the ordered pair (T_d^{SIMD}, T_d^{SPMD}), which represents the times for executing the data-conditional construct at node d, including the time to execute all the children of d in SIMD mode and SPMD mode, respectively.

To estimate the values of the ordered pair (T_d^{SIMD}, T_d^{SPMD}), consider how data conditionals are performed in SIMD and SPMD modes.

In SIMD mode, if the conditional test for an `if-then-else` is true for *all* PEs, then only the PE instructions that belong to the `then` clause need to be broadcast to the PEs. Similarly, if the conditional test is false for *all* PEs, only the `else` clause needs to be broadcast. However, if the condition is true for some PEs and false for others, then the PE instructions for both the `then` and `else` clauses must be broadcast, effectively serializing the two clauses. Let p_{then} denote the probability that a PE executes the `then` clause, let P_{then} denote the probability that *all* PEs execute the `then` clause, and let P_{else} denote the probability that *all* PEs execute the `else` clause. Estimates for the values of P_{then}, P_{else}, and p_{then} can be determined in a similar manner to the value of Q as discussed in Subsection 3.1 (e.g., empirical data and/or compiler directives). Let T_{then}^{SIMD} and T_{else}^{SIMD} be the sum of the execution times of all the children of the `then` and `else` clauses in SIMD mode, respectively. Then the expected time required to perform the `if-then-else` in SIMD mode can be estimated by

$$T_d^{SIMD} = P_{then} T_{then}^{SIMD} + P_{else} T_{else}^{SIMD} + (1 - P_{then} - P_{else})(T_{then}^{SIMD} + T_{else}^{SIMD})$$

$$= (1-P_{else})T_{then}^{SIMD} + (1-P_{then})T_{else}^{SIMD} .$$

In SPMD mode, each PE independently follows its own control path through the program. The `then` clause is executed on those PEs where the condition is true, and the `else` clause is executed on those PEs where the condition is false. Let T_{then}^{SPMD} and T_{else}^{SPMD} be the sum of the execution times of all the children of the `then` and `else` clauses in SPMD mode, respectively. Then the expected time to perform the conditional construct in SPMD mode is

$$T_d^{SPMD} = p_{then} T_{then}^{SPMD} + (1 - p_{then}) T_{else}^{SPMD} .$$

3) For each non-leaf node c corresponding to a *control* construct, assign the ordered pair (T_c^{SIMD}, T_c^{SPMD}), which represents the times for executing the control construct at node c, including the time to execute all the children of c in SIMD mode and SPMD mode, respectively.

In SPMD mode, the control construct is performed entirely on the PEs. In SIMD mode, the control steps are performed on the CU, and then the PE instructions that form the loop body are placed in the instruction queue to be broadcast to the PEs. For SIMD, significant improvement in execution time can be obtained by overlapping the execution of operations on the CU and the PEs, and is discussed in the present context in [29]. Disregarding the effects of CU/PE overlap, the ordered pair (T_c^{SIMD}, T_c^{SPMD}) can be estimated by

$$T_c^{SIMD} = Q\left[\sum_{\substack{\text{all children} \\ u \text{ of } c}} T_u^{SIMD}\right], \quad T_c^{SPMD} = Q\left[\sum_{\substack{\text{all children} \\ u \text{ of } c}} T_u^{SPMD}\right],$$

where Q is the number of iterations in the construct and is assumed to be known, as discussed in Subsection 3.1.

4) For the node representing the root, assign the ordered pair $(T_{root}^{SIMD}, T_{root}^{SPMD})$, which is the sum over all the children of the root, and represents the times for executing the entire program in SIMD mode and SPMD mode, respectively.

If $T_{root}^{SPMD} \le T_{root}^{SIMD}$, then implement the algorithm in SPMD mode, else implement it in SIMD mode.

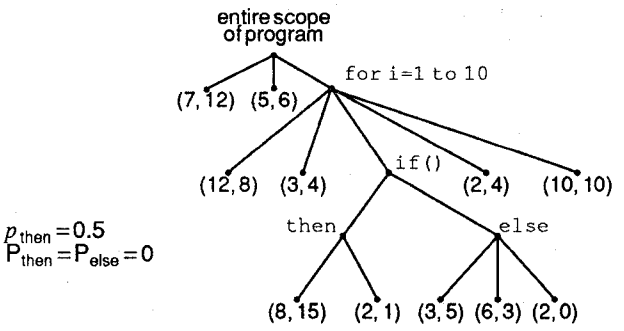

Fig. 3: Trade-off tree for the final flow-analysis tree in Fig. 2.

As an example, consider the SIMD/SPMD trade-off tree in Fig. 3, which corresponds to the the flow-analysis tree in Fig. 2. In the tree, the ordered pairs (T_l^{SIMD}, T_l^{SPMD}) are assigned to each of the leaf nodes. For the example, Q = 10, $p_{then} = 0.5$, and let $P_{then} = P_{else} = 0$. The analysis algorithm performs a depth-first traversal beginning at the root node. When the algorithm considers the `then` clause, it assigns the ordered pair $(T_{then}^{SIMD}, T_{then}^{SPMD}) = (8+2, 15+1) = (10, 16)$. Similarly, for the *else* clause, $(T_{else}^{SIMD}, T_{else}^{SPMD}) =$

(3+6+2, 5+3+0)= (11, 8). Then the algorithm considers the `if` node, and assigns the ordered pair (T_d^{SIMD}, T_d^{SPMD}) = (10+11, 0.5×16+0.5×8)= (21, 12). At the `for` node, the algorithm assigns the ordered pair (T_c^{SIMD}, T_c^{SPMD}) = (10×(12+3+21+2+10), 10×(8+4+12+4+10)) = (480, 380). For the root node, the ordered pair (7+5+480, 12+6+380) = (492, 398) is assigned, indicating that SPMD mode is best suited for this program.

4: Mixed-mode and mixed-machine analysis

4.1: Assumptions and definitions

In mixed-mode and mixed-machine systems, it is possible for some portions of a single parallel program to be implemented in SIMD mode, and other portions of the same program are executed in SPMD mode. This will be referred to as a mixed-algorithm. The analysis of single-mode parallel programs developed in Section 3 is expanded here to consider mixed-algorithms.

For the analysis here, several simplifying assumptions are made on the selection of modes within a program. Leaf blocks are assumed to be implemented completely in either SIMD or SPMD mode. Given the definition of a block, such that control and data-conditional constructs, synchronizations, inter-PE transfers, and CU operations all determine the boundaries of a block, there is no benefit in changing modes within a leaf block.

Another assumption is that if a block is to be executed more than once, i.e., as part of a looping construct, the mode of parallelism for that block is the same for all iterations. Because there are no branches or targets of branches within a block, and no inter-PE transfers or synchronization points within a block, there is no perceived advantage for the same block to execute in different modes from iteration to iteration; i.e., if a given mode is better for one instance of a block, it should be better for all instances.

Additionally, all blocks that comprise each data-conditional construct are to be implemented in the same mode of parallelism, i.e., for each `if` construct, all the blocks within the construct are either implemented in SIMD mode, or they are all implemented in SPMD mode. This assumption is necessary because mode changes within data-conditional constructs can translate into operations that cannot be implemented without excessive execution-time overhead [29].

Finally, it is assumed that each iteration of a loop must begin and end execution in the same mode of parallelism. Consider, for example, a sequence of blocks that comprise the body of a loop, such that the first block is implemented in SIMD mode and the last block is implemented in SPMD mode. Between each successive iteration of the loop, a mode-switch from SPMD to SIMD is required to allow the PEs to perform the first block of the next iteration. Because the last block does not end in the same mode as the first block, a mode switch is added at the beginning of the loop body.

4.2: Optimal mixed-algorithm selection

To perform the execution time analysis for a mixed-algorithm, a SIMD/SPMD trade-off tree is constructed where, as before, each block in the program is a leaf node and the data-conditional and control constructs form the interior nodes, with the root node representing the scope of the entire program. For mixed-algorithms, the mode of parallelism may be changed between adjacent sibling nodes, with an appropriate time cost. These mode changes are not represented in the SIMD/SPMD trade-off tree.

For the following discussion, let the ordered pair ($T^{SIMD}_{mixed\text{-}n}$, $T^{SPMD}_{mixed\text{-}n}$) represent the minimum mixed-algorithm execution time estimate for a node n, where n is not the root node for the entire program, i.e., for a subtree with root n that begins and terminates execution in SIMD or SPMD, respectively, but may switch modes zero or more times during execution. The values for $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ include the time required for any mode switches that are performed. In the SIMD/SPMD trade-off tree, interior nodes can represent `if` constructs, `then` and `else` clauses, and `for` constructs. Recall that for non-leaf nodes corresponding to descendents of data-conditional constructs, mode changes are disallowed. Therefore, for nodes corresponding to `if`, `then`, and `else` nodes, $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ represent the estimated execution time for that node purely in SIMD or purely in SPMD. Recall that for a sequence of nodes that are children of a `for` 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. Therefore, the mode of the `for` subtree is defined to be that of the last node that is a child of that `for` 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.

There is a cost C^{SIMD} associated with switching to SIMD mode, and a cost C^{SPMD} associated with switching to SPMD mode. The values of C^{SIMD} and C^{SPMD} are assumed to be known constants. It is expected that for the mixed-machine case, C^{SIMD} and C^{SPMD} will be greater than for a single mixed-mode machine, because data may have to be moved between machines. The application of this technique under the relaxed assumption of non-constant mode-switching costs for mixed machines is currently under study.

The following steps are performed to determine the execution times for all nodes:

1) Same as in Subsection 3.2.

Steps 2 and 3 are then performed for each non-leaf node in the order of a depth-first traversal of the tree:

2) Same as in Subsection 3.2.

Because of the single execution mode constraint assumed for data-conditional constructs, $T^{SIMD}_{mixed\text{-}d} = T^{SIMD}_d$ and $T^{SPMD}_{mixed\text{-}d} = T^{SPMD}_d$.

3) Case (a): For each non-leaf node c corresponding to a control construct that is not in a subtree with a data-conditional construct as its root, assign an ordered pair ($T^{SIMD}_{mixed\text{-}c}$, $T^{SPMD}_{mixed\text{-}c}$), which represents the times for executing the control construct at node c, beginning and ending in SIMD and SPMD modes, respectively, including the time to execute all the children of c.

Let the ordered pair ($T^{SIMD}_{mixed\text{-}iteration}$, $T^{SPMD}_{mixed\text{-}iteration}$) represent the minimum mixed-algorithm execution time estimate for a

single iteration of the loop corresponding to node c. Because the last block in the loop is the `for_test`, and because the first and the last block must be in the same mode of parallelism, $T^{SIMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SIMD on the CU, and $T^{SPMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SPMD on the PEs. Recall that Q is the number of iterations of the loop to be executed. Then $(T^{SIMD}_{mixed\text{-}c}, T^{SPMD}_{mixed\text{-}c})$, is given by

$$T^{SIMD}_{c} = Q \times T^{SIMD}_{mixed\text{-}iteration}, \; T^{SPMD}_{c} = Q \times T^{SPMD}_{mixed\text{-}iteration}.$$

Recall that if the first node of a loop body and the `for_test` are in different modes, a mode switch is inserted before the first node in the loop body. This may lead to an unneeded mode switch for the first iteration. This situation is described in detail later.

Case (b): If node c is the descendent of a node corresponding to an `if` construct, then only pure SIMD and pure SPMD implementations are considered, and the equations for the single-mode analysis methodology are used instead.

4) For the node representing the root, assign the ordered quadruple $(T^{SIMD/SIMD}_{root}, T^{SIMD/SPMD}_{root}, T^{SPMD/SIMD}_{root}, T^{SPMD/SPMD}_{root})$, where $T^{X/Y}_{root}$ corresponds to the minimum mixed-algorithm execution time required to perform all the children of the root node, beginning in mode X and ending in mode Y.

Thus, it is possible for the root node to avoid a final mode change. This is desirable for mixed-machine implementations, where C^{SIMD} and C^{SPMD} are generally large. The minimum quadruple value then represents the best mixed-mode or mixed-machine implementation.

As the SIMD/SPMD trade-off tree is traversed, the deepest levels of the tree are combined by employing the above steps. As the analysis works its way up the tree, higher levels are combined, until only the root is represented. Then the parallel mode for each segment of the program can be assigned. A numerical example for the method described here is illustrated in [29].

One effect that must be considered when traversing the tree is the possibility of adding unnecessary mode switches when going from the `for_init` block to the first child in the loop body of a `for` node. For example, consider a `for` construct such that the `for_init` block is implemented in SPMD, the first child in the loop body of the `for` node is implemented in SPMD, and the `for_test` block (the last child of the `for` node) is implemented in SIMD, as illustrated in Fig. 4a. Because the last and first nodes are performed one after the other when iterating over the loop, a mode switch from SIMD to SPMD is required before executing the first node. Additionally, by the definition of a `for` loop node, the mode of the `for` node is the same as that of the `for_test` in the loop. Thus, a mode switch from the SPMD `for_init` node to the SIMD `for` node is inserted before the first iteration of the loop body. However, immediately after the execution of the `for_init`, and before the first iteration, the parallel system is already in SPMD mode, which is the mode of parallelism required for the first node in the loop body. By detecting this case, the mode switch from

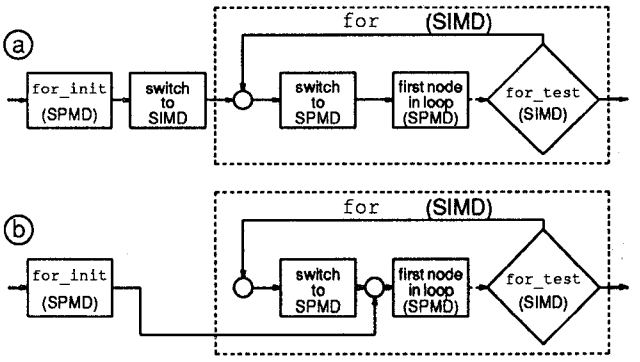

Fig. 4: Modification of `for` loop to avoid unnecessary mode switches for the first iteration.

SPMD to SIMD and from SIMD back to SPMD can be avoided for the first iteration, as illustrated in Fig. 4b.

4.3: Computational aspects

Exhaustively testing all possible implementations to find the minimum mixed-algorithm execution time for a node with m children would require testing 2^m combinations. For nodes with many children, this approach is impractical; therefore, an efficient method of finding the minimum execution time is needed when m becomes large.

An efficient way of computing $(T^{SIMD}_{mixed\text{-}n}, T^{SPMD}_{mixed\text{-}n})$ is to transform the subtree rooted at 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 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, 9].

Multistage optimization problems can be solved using Moore's algorithm [20] (a variant of Dijkstra's algorithm [12]) 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.

An example of a multistage optimization problem and its solution is illustrated in Fig. 5. Initially, there are 4 stages, numbered from 0 to 3. In the first iteration, for each vertex g in stage 0, a minimum path is found from g to each vertex h 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 (g,h) pair. The multistage optimization algorithm 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 problem resulting in the minimum cost for each initial/final pair is obtained.

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 to a single node. Let each mode of parallelism represent a separate state within each stage. Let the cost associated with each edge correspond to the cost of performing that node in SIMD mode or SPMD mode. Then, before the representation of each node in the multistage graph, include a stage and associated edges representing the cost of a possible mode switch between nodes.

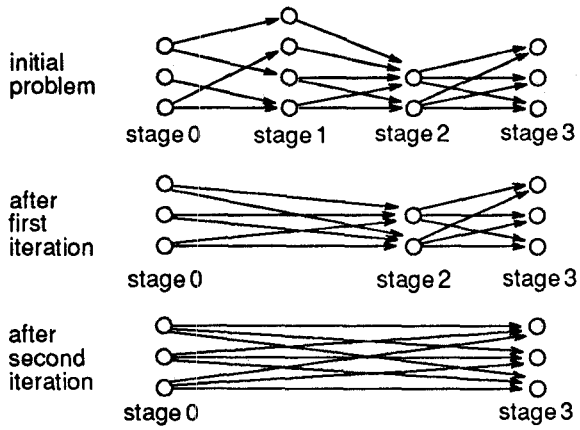

Fig. 5: Example of multistage optimization problem and its solution.

As an example, consider the transformation illustrated in Fig. 6a. 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 Fig. 6a) 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, while 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 Fig. 6a represent the cost of executing the node in each mode. Fig. 6b illustrates the graph for a SIMD/SPMD trade-off tree with three children. The ordered pair $(T^{SIMD}_{mixed\text{-}4}, T^{SPMD}_{mixed\text{-}4})$ is obtained from the solution of the multistage optimization problem.

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 find 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.

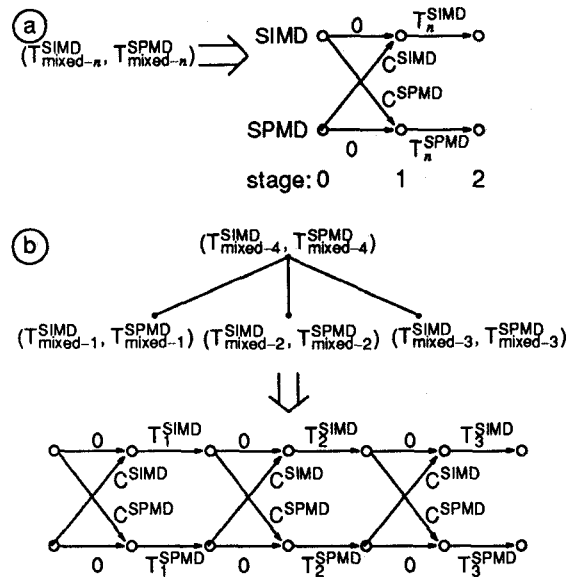

Fig. 6: Transformation from flow-analysis tree to multistage graph.

By recording the minimum paths selected at each iteration, the mode of parallelism for each stage is selected. 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, 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 up the trade-off tree until execution costs are obtained for the root node.

5: Summary and future work

The block-based mode selection model proposed establishes a framework on which to build analysis techniques for compile-time selection of parallel modes in an SIMD/SPMD heterogeneous context. The model consists of an algorithm for determining information in an SIMD/SPMD trade-off tree, which is then combined using a set of rules and by employing a multistage optimization technique to determine the minimum mixed-algorithm execution time for a sequence of nodes. With this approach, parallel programs written in a mode-independent language can be mapped onto the architecture and/or mode of parallelism most appropriate for that program or program segment.

There are various extensions to the model that form the basis for future work. The set of control and data-conditional constructs can be expanded to include other useful constructs, e.g. `while` statements, `case` statements,

and function calls. The probabilistic model discussed in [29] can be enhanced to consider more fully the effect of juxtaposed blocks on overall execution time. The framework can also include a more complete model of CU/PE overlap than used in [29]. Other research may involve more practical aspects of the analysis, for example, the details of incorporating the techniques presented here into a parallel compiler. Research studies directed toward obtaining the statistical information assumed in this work would provide information necessary to implement the model.

For a practical implementation in a mixed-mode environment, the time to move between machines will not be a constant for all blocks, and decisions at one point will impact future data movements. Therefore, an important extension to this study is to examine the case where the cost of switching machines is not constant, but depends on the size, location, and usage of data within the program.

Another research area that is the subject of future work is the impact on the analysis of including computation models other than SIMD and SPMD, e.g., MIMD and vector processing. The incorporation of other parallel/vector models would form the basis for future efforts to provide compiling tools for heterogeneous systems.

Acknowledgments: The authors wish to thank J. Armstrong, R. Born, T. Casavant, H. Dietz, A. Maciejewski, W. Nation, R. Palmer, G. Saghi, R. Ulrey, and R. Wolski for their valuable comments.

References

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

[2] J. Antonio, W. Tsai, and G. 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] M. Auguin and F. Boeri, "The OPSILA computer," in *Parallel Languages and Architectures,* M. Consard, ed., Elsevier Science Publishers, Holland, 1986, pp. 143-153.

[4] M. Auguin and F. Boeri, "Experiments on a parallel SIMD/SPMD architecture and its programming," *France-Japan Artificial Intelligence and Computer Science Symp. 87,* Nov. 1987, pp. 385-411.

[5] T. Berg, S. Kim, and H. J. Siegel, "Limitations imposed on mixed-mode performance of optimized phases due to temporal juxtaposition," *J. Parallel and Distributed Computing,* Vol. 13, No. 2, Oct. 1991, pp. 154-169.

[6] T. Berg and H. J. Siegel, "Instruction execution trade-offs for SIMD vs. MIMD vs. mixed-mode parallelism," *5th Int'l Parallel Processing Symp.,* May 1991, pp. 301-308.

[7] E. Bronson, T. Casavant, and L. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 2, Apr. 1990, pp. 195-205.

[8] J. Brown, M. Azam, and S. Sobec, "CODE: A unified approach to parallel programming," *IEEE Software,* July 1989, pp. 10-17.

[9] A. Bryson and Y. Ho, *Applied Optimal Control,* Hempshire, Washington, DC, 1975.

[10] H. Dietz and D. Klappholz, "Refined C: a sequential language for parallel programming," *1985 Int'l Conf. Parallel Processing,* Aug. 1985, pp. 442-449.

[11] H. Dietz, A. Zaafrani, and M. O'Keefe, "Static scheduling for barrier MIMD architectures," *J. Supercomputing,* Vol. 5, 1992, pp. 263-289.

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

[13] S. Fineberg, T. Casavant, and H. J. Siegel, "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting," *J. Parallel and Distributed Computing,* Vol. 11, No. 3, Mar. 1991, pp. 239-251.

[14] R. Freund, "Optimal selection theory for superconcurrency," *Supercomputing '89,* Nov. 1989, pp. 699-703.

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

[16] N. Giolmas, D. Watson, D. Chelberg, and H. J. Siegel, "A parallel approach to hybrid range image segmentation," *6th Int'l Parallel Processing Symp.,* Mar. 1992, pp. 334-342.

[17] M. Gupta and P. Banerjee, "Compile-time estimation of communication costs on multicomputers," *6th Int'l Parallel Processing Symp.,* Mar. 1992, pp. 470-475.

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

[19] L. 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.

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

[21] M. Nichols, H. J. Siegel, and H. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler," *IEEE Trans. Parallel and Distributed Systems,* Vol. 4, No. 1, Jan. 1993, to appear.

[22] M. Philippsen, T. Warschko, W. 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.

[23] B. Qin, H. Sholl and R. Ammar, "Micro time cost analysis of parallel computations," *IEEE Trans. Computers,* Vol. 40, No. 5, May 1991, pp. 613-628.

[24] H. J. Siegel and S. Abraham, et al., "Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing," *J. Parallel and Distributed Computing,* Vol. 16, No. 3, Nov. 1992, pp. 199-211.

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

[26] V. Sunderam, "Design issues in heterogeneous network computing," revised edition of proceedings, *Workshop on Heterogeneous Processing,* May 1992, pp. 101-112.

[27] L. Tucker and G. Robertson, "Architecture and applications of the Connection Machine," *Computer,* Vol. 21, No. 8, Aug. 1988, pp. 26-38.

[28] M. Wang, S. Kim, M. Nichols, R. Freund, H. J. Siegel, and W. Nation, "Augmenting the optimal selection theory for superconcurrency," *Workshop on Heterogeneous Processing,* May 1992, pp. 13-21.

[29] D. Watson, H. J. Siegel, J. Antonio, M. Nichols, and M. Atallah, *A Block-Based Approach for Selecting Modes in a SIMD/SPMD Heterogeneous Environment,* technical report under preparation, School of Electrical Engineering, Purdue University, Jan. 1993.

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