Efficient Emulation of MIMD Behavior on SIMD Machines

Peter Sanders
LS Informatik für Ingenieure und Naturwissenschaftler
Universität Karlsruhe
D-76128 Karlsruhe
Germany
E-mail: `sanders@ira.uka.de`
Fax: (49) 721-698 675

Abstract

SIMD computers have proved to be a useful and cost effective approach to massively parallel computation. On the other hand, there are algorithms which are very inefficient when directly translated into a data-parallel program. This paper presents a number of simple transformations which are able to reduce this *SIMD overhead* to a moderate constant factor. It also introduces techniques for reducing the remaining overhead using Markov chain models of control flow. The optimization problems involved are **NP**-hard in general but there are many useful heuristics, and closed form optimizations for a probabilistic variant.

Keywords: Markov chain, **NP**, optimization, parallel search, SIMD MIMD emulation.

1 Introduction

Single Instruction Multiple Data computers are a quite effective approach to massively parallel computation. Since instructions are stored and decoded centrally, the processing elements (PEs) do not need instruction memory or a control unit. This makes it feasible to integrate many simple PEs on an area normally required for one single processor of a MIMD machine. In addition, the synchronous nature of SIMD processing often makes programming easier.

On the other hand, there are algorithms which are very inefficient when directly translated into a data-parallel program. This happens whenever there is a loop with a small number of iterations on most PEs for which there is at least one PE with a much larger number of iterations. In this case all PEs have to wait until the last PE finishes. For programs whose execution time is dominated by such loops, a SIMD computer performs very poorly. A simple example is the task to compute the Mandelbrot set [15]; the number of iterations necessary to compute a point varies widely. In [6] methods for restructuring nested loops which are typical for numerical applications are discussed. Even more complex control flow patterns can be observed for irregular nonnumeric applications. In the following, depth-first tree search is used as an example.

```
Let $s$ be a stack containing only the root node
of a subtree to be processed on this processor
LOOP
    IF isLeaf(top(s)) THEN
        IF isSolution(top(s)) THEN printSolution(top(s))
        WHILE noMoreSiblings(top(s)) DO
            pop(s)
            IF isEmpty(s) THEN stop
        replace top(s) with nextSibling(top(s))
    ELSE push(firstSuccessor(top(s)))
```

Figure 1: Nonrecursive generic depth-first search.

The pseudocode in Figure 1 gives the kernel of a nonrecursive, parallel depth-first search algorithm which searches for leaves constituting a solution. It is assumed that a load balancer takes care that each PE gets a different subtree of the search space. A subtree is represented by its root. Interior nodes are expanded by pushing their first successor on the stack. For leaf nodes, it is checked whether they constitute a solution. Then the program backtracks to the next node with unsearched siblings.

For many problems, the while-loop responsible for backtracking performs very few iterations on the average; but on some PEs the number of iterations may be the full depth of the tree. Additional complications could be introduced by heuristics, or by loops inside the application-specific functions isLeaf, firstSuccessor, etc.

This paper presents techniques for removing the offending loops and producing an equivalent, more efficient program. It is an extended version of the conference paper [13]. Section 2 presents the basic techniques and a number of ideas for optimization. Then Section 3 models control flow of programs using Markov chains. The results make it possible to find optimizations with less trial and error. In Section 4 these techniques are applied to a heuristic search problem raised by an open question in cellular automata theory. Section 5 discusses a different approach to transforming programs into synchronous form which is used by many other researchers. It turns out that it can be viewed as a special case of our transformation approach and that the Markov chain models again yields interesting insights. It is also proved that even for this special case, one of the basic optimization problems considered is **NP**-hard. Finally, Section 6 summarizes the results.

2 Transformation into synchronous form

```
initialization
LOOP
    IF $g_1$ THEN $o_1$
    IF $g_2$ THEN $o_2$
    ⋮
    IF $g_n$ THEN $o_n$
```

Figure 2: Test loop suitable for a SIMD machine.

The general idea for "SIMD-izing" an algorithm is very simple. Every MIMD program can be transformed into a SIMD program of the general form of a *test loop* given in Figure 2.

The o_i are *elementary operations* (still to be determined) which do not contain loops. (More precisely, loops with a globally known number of iterations are no problem.) The statement **IF** g_i **THEN** o_i is a *test* for operation o_i. More generally, for many applications it makes sense to decompose the program (possibly dynamically) into a sequence of test loops — each with a different arrangement of tests — but, since the loops can be investigated one at a time, we can restrict ourselves to one loop.

If the control logic of an algorithm can be implemented by a finite automaton then the test loop can be constructed by introducing one elementary operation for each state. For example, Algorithm 1 can be transformed into the test loop depicted in Figure 3.

The key observation is that every problem can be cast into this shape:

1. Without loss of generality assume that all PEs run the same process (Single Program Multiple Data programming model).

2. Eliminate calls to procedures which contain loops with a varying number of iterations. This can be done by inlining or by replacing procedure calls by appropriate stack manipulations and control structures.

3. Implement loop control by goto-statements.

4. The code sections between goto-labels now constitute the set of elementary operations; i.e. a code section label: code is replaced with
IF state = LABEL **THEN** code and a jump goto label is replaced with the assignment state := LABEL. (The labels are replaced with unique constants.)

This transformation could, for example, be performed by a compiler. For manual use however, it is better to step back and select states which have a meaningful interpretation in the application domain.

Note that there are two different kinds of control flow. One is the control flow of the problem to be emulated. We call this the asynchronous control flow or simply control flow. The other is the control flow of the test loop which

```
initialize as in Algorithm 1
state := Search
LOOP
    IF state = Search THEN
        IF isLeaf(top(s)) THEN
            IF isSolution(top(s)) THEN state := Solution
            ELSE state := GetNextChoice
        ELSE state := MakeChoicePoint
    IF state = MakeChoicePoint THEN
        push(firstSuccessor(top(s))); state := Search
    IF state = GetNextChoice THEN
        IF noMoreSiblings(top(s)) THEN
            pop(s)
            IF isEmpty(s) THEN stop
            state := GetNextChoice
        ELSE top(s) := nextSibling(top(s)); state := Search
    IF state = Solution THEN
        printSolution(top(s)); state := GetNextChoice
```

Figure 3: Depth-first search controlled by an automaton.

deterministically cycles through the tests. Here we talk about the position in the test loop.

A test loop along the pattern of Figure 2 still contains two sources of inefficiency: First, the required number of iterations through the outer loop can vary from PE to PE. But this problem of *load imbalance* is a general problem of parallel computing which also occurs on MIMD computers. Therefore, load balancing strategies are not discussed here. Furthermore, during a test for an operation o_i, all PEs for which g_i does not hold, are deactivated. (We call this an *unproductive test.*) This remaining *SIMD overhead* depends on the complexity of the program and not on the problem size. Therefore, every MIMD program can be emulated by a SIMD program with constant overhead. Still, in practice it is important to keep this constant small in order to be competitive with MIMD machines.

2.1 Optimizing the test loop

So far, we have always considered test loops which test for every operation exactly once in some arbitrary order. But we are free to select any order of tests. We can even duplicate tests if this helps. As a general heuristics, it is a good idea to test for cheap, frequently needed operations more often than for expensive, rarely needed ones. Also, the tests should be ordered in such a way that a maximum total number of productive tests per iteration of the test loop is performed.

For example, let us assume that interior nodes of the trees to be traversed by Algorithm 3 have many descendents, that there are very few solutions and that operation Solution takes 10 units (of time) while all other operations cost 1 unit. The control flow is therefore dominated by subsequences of the form Search; GetNextChoice; Search; GetNextChoice; ... The test loop Search; GetNextChoice; MakeChoicepoint; Solution takes 13 units and about 2 productive tests per iteration are performed. The test loop Search; GetNextChoice; Search; GetNextChoice; MakeChoicepoint; Solution on the other hand, takes 15 units but about 4 productive tests per iteration are performed — its almost two times more efficient.

Similar ideas are discussed in [3, 1, 10]. In [10] it is argued that duplicating tests is useless since some PEs are actually delayed due to large deviations from the average control flow. But this is not always a problem. Often all PEs have quite similar control flow characteristics, and even if there are PEs which are delayed, this only means that operation duplication has increased *load imbalance* which only results in a longer *execution time* if the load balancer is not able to cope with the additional imbalance. Depth-first tree search for example, can be a very irregular problem anyway and a load balancer which works for a simple test loop has no trouble handling the minor additional imbalance from test duplication.

2.2 Selecting Operations

So far, we have assumed that the set of operations is fixed. However, there are a number of useful transformations on operations which help to increase efficiency:

Splitting: An operation of the form

α
IF c **THEN** β
γ

can be replaced with the following three operations:

o_α: α; state := (**IF** c **THEN** o_β **ELSE** o_γ)
o_β: β; state := o_γ
o_γ: γ

This is useful if the branch β is rarely taken or very expensive. Since β is an operation of its own now, it can be tested for less frequently than other cheaper or more important operations. For the case discussed in Section 2.1 for example, it is a good thing to have an independent operation Solution instead of making it part of the operation Search. Still, sometimes the inverse operation of incorporating an operation into another is also useful in order to decrease control overhead. It should have become clear now that it is not clear at all when to apply what transformation. This is the reason why Sections 3 and 5 develop mathematical models which help to make these decisions.

Simplification:

In traditional programs, most code need not be fine-tuned since only the small fraction of code in the inner loop is critical. In our approach the *entire* test loop is the inner loop. So, tuning rarely used operations can have an unexpected impact on performance. On the other hand, traditional programs often profits from optimized treatment of some special cases. In a SIMD program however, this approach may backfire since the code for the special case incurs additional

SIMD overhead. In a sense, *removing* optimizations is sometimes the better optimization.

Merging: If two operations are almost identical like

o_1: α; $\mathsf{state} := o'_1$
o_2: α; $\mathsf{state} := o'_2$

they can be merged into the single operation

o_{12}: α; $\mathsf{state} := \mathsf{follow}$

if other operations assign the proper value to follow before setting state to o_{12}. This transformation reduces the number of operations and therefore decreases SIMD overhead. Often, splitting and simplification of operations can be used to produce candidates for merging. Essentially, merging is a primitive kind of procedure call and the idea can be expanded to nested calls and recursion by introducing a return stack. In [3] a method called *common subexpression induction* is mentioned which automatically recognizes mergeable parts of code.

3 Modeling control flow with Markov chains

It can involve a lot of trial and error to apply the optimizations in Sections 2.1 and 2.2. Therefore, this section develops mathematical tools which help to select appropriate transformations.

The first step is to abstract from the problem of load balancing which is application dependent and not a specific problem of SIMD computing. This can conveniently be done by assuming infinite load on every processor. Performance is then naturally expressed as the average number of productive tests per unit of time (throughput). The choice of the next operation depends on the current operation and some unknown (hidden) computation we assume to be random. Under these assumption the operations can be identified with the states of a Markov chain. Let p_{ij} designate the transition probability i.e. the probability that the operation o_i follows o_j in the asynchronous control flow. Let c_i be the cost of testing for operation o_i. This model was developed independently from [10] where it is used in a slightly different and simplified setting.

3.1 Assessing the performance of a test loop

Using the Markov chain model above it is possible to predict the performance of a candidate test loop. This can be done using a kind of symbolic execution: Given the transition probabilities and a vector containing the probabilities that the asynchronous control flow is currently in a given state, it is possible to compute the impact of the next test on this vector. By keeping track of the cost of tests and the fraction of PEs which do productive tests and by iterating a few times through the test loop, a cost function expressing the average cost per productive test can be approximated. (For details refer to [12].)

This is equivalent to modeling asynchronous control flow *and* a specific test loop by a Markov chain: A state s_{ik} represents a situation where o_i is the operation needed next and k is the current position in the test loop. Using this approach, the cost function can be computed by solving a large, sparse eigenvector equation. The above symbolic execution approach can be viewed as an iterative solver for this eigenvalue equation which implicitly exploits the sparseness of the transition matrix.

Either way, we now have a tool for quickly screening a number of alternative test loops without having to run the program once the parameters p_{ij} and c_i have been measured for typical input data. Unfortunately, the task of finding an optimal test loop for a given control flow turns out to be **NP**-hard. In order to make this more precise we formulate our problem as a decision problem in the format of [4]:

TEST LOOP SCHEDULING

INSTANCE: Operations $o_1, \ldots, o_n$ and costs $c_i \in \mathbf{N_0}$; transition probabilities $p_{ij} \in \mathbf{Q}$, a test loop length m and a cost bound $C \in \mathbf{Q}^+$.

QUESTION: Is there a test loop of length m for which the Markov chain model predicts a cost $C' \leq C$?

TEST LOOP SCHEDULING is **NP**-hard because it is a generalization of the SUBINTERPRETER SCHEDULING problem to be discussed in Section 5.2. Even for moderate numbers of operations we therefore have to resort to heuristics like hill climbing or genetic algorithms in order to arrive at good test loops.

3.2 Optimal probabilistic test loops

Since assembling optimal test loops turns out to be intractable, it is a logical idea to further simplify the model in order to be able to derive closed form results. Therefore, we now abstract from the execution order in the asynchronous control flow and the test loop: We assume that operations can be fully characterized by their frequency p_i of occurrence in the asynchronous control flow. (It can be measured by counting how often an operation is actually executed.) Now we want to know at which frequency f_i operation o_i should be tested for in order to achieve optimal throughput. This frequency is defined by a probabilistic test loop which randomly decides which operation is tested for next:

LOOP choose i with probability f_i; **IF** g_i **THEN** o_i **ENDLOOP**

For this program, we can define a cost function $C(f_1, \ldots, f_n)$ which gives the average cost for performing one productive test. Stated probabilistically, this means the ratio between the expected time $\sum f_i c_i$ for executing a test, and the probability a of a productive test.

$$C(f_1, \ldots, f_n) := \frac{\sum_{i=1}^{n} f_i c_i}{a(f_1, \ldots, f_n)} \tag{1}$$

In order to determine a, the asynchronous control flow is modeled by a Markov chain with the states A for "active" and W_j for "waiting for a test for o_j". Figure 4 shows the states and the transition probabilities. If the control flow is waiting for a test for o_j, this operation is tested for next with a probability of f_j and the Markov chain makes a transition to the active state, else waiting continues. If the

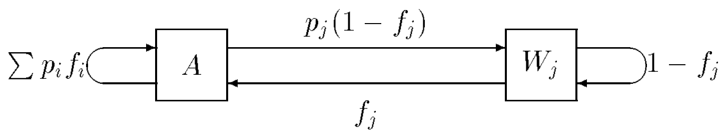

Figure 4: Markov model of the asynchronous control flow ($(n+1)$ states).

Markov chain is in the active state and the next operation in the asynchronous control flow is o_j but o_j is not tested for next, there is a transition to the waiting state W_j. Summing probabilities for the remaining cases yields a probability of $\sum p_i f_i$ for a transition from active to active.

Now, the $a(f_1, \ldots, f_n)$ we are looking for, is the equilibrium probability to find the Markov chain in the active state. This probability can be obtained by solving the eigenvalue equation

$$\begin{pmatrix} \sum p_i f_i & f_1 & \cdots & f_n \\ p_1(1-f_1) & 1-f_1 & & \mathbf{0} \\ \vdots & & \ddots & \\ p_n(1-f_n) & \mathbf{0} & & 1-f_n \end{pmatrix} \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \tag{2}$$

which corresponds to the following homogeneous linear equation system:

$$\begin{pmatrix} (\sum p_i f_i) - 1 & f_1 & \cdots & f_n \\ p_1(1-f_1) & -f_1 & & \mathbf{0} \\ \vdots & & \ddots & \\ p_n(1-f_n) & \mathbf{0} & & -f_n \end{pmatrix} \begin{pmatrix} a \\ w_1 \\ \vdots \\ w_n \end{pmatrix} = 0$$

Adding all rows eliminates the top row. After dividing row i by f_i we have

$$a \frac{p_i(1-f_i)}{f_i} = w_i.$$

Adding these equations and using the additional condition $a + \sum w_i = 1$ yields

$$a \sum \frac{p_i(1-f_i)}{f_i} = 1 - a \text{ or } a = \frac{1}{\sum_{i=1}^{n} \frac{p_i}{f_i}}. \tag{3}$$

Substituting this result into Equation 1 completes the cost function.

$$C(f_1, \ldots, f_n) = \left(\sum_{i=1}^{n} \frac{p_i}{f_i} \right) \left(\sum_{i=1}^{n} f_i c_i \right) \tag{4}$$

This equation can be used to determine the optimal testing frequencies f_i by looking for roots of the partial derivatives

$$\frac{\partial C}{\partial f_k} = -\frac{p_k}{f_k^2} \sum_i c_i f_i + \left(\sum_i \frac{p_i}{f_i} \right) c_k \overset{!}{=} 0.$$

Moving the k-dependent terms to the left yields

$$\frac{c_k f_k^2}{p_k} = \frac{\sum c_i f_i}{\sum \frac{p_i}{f_i}}.$$

Equating two instances of these equations removes the sums

$$\frac{c_k f_k^2}{p_k} = \frac{c_j f_j^2}{p_j} \text{ or } \frac{f_k}{f_j} = \frac{\sqrt{\frac{p_k}{c_k}}}{\sqrt{\frac{p_j}{c_j}}},$$

and introducing the additional condition $\sum f_k = 1$ gives the result

$$f_j = \frac{\sqrt{\frac{p_j}{c_j}}}{\sum_{i=1}^{n} \sqrt{\frac{p_i}{c_i}}}. \tag{5}$$

Since this is the only candidate for an extremal point and since C approaches infinity if any of the f_i approaches zero or one, this solution constitutes the global optimum.

The potential speedup due to unequal testing frequencies can be estimated by substituting the optimal testing frequencies from Equation 5 and the straightforward case $f_i = 1/n$ into the cost function from Equation 4.

$$S = \frac{C_{\text{naive}}}{C_{\text{opt}}} = \frac{\sum_{i=1}^{n} c_i}{\left(\sum_{i=1}^{n} \sqrt{p_i c_i}\right)^2} \tag{6}$$

The value for C_{naive} also indicates that it is not a good idea to use a probabilistic test loop for the actual implementation because on the average every operation has to be tested for once before the right one is found. For a simple deterministic test loop which tests for every operation once this is the absolute worst case; it is more likely that testing for about half the operations is enough on the average. Nevertheless, we can expect that the frequencies computed for the probabilistic case are also useful for a deterministic implementation. The simple formulas can be used to quickly estimate the impact of one of the operation transformations in Section 2.2 or to construct initial test loops as a starting point for manual or automatic iterative improvement (refer to Section 5.3 for an example) and they can be used to roughly estimate the SIMD overhead and its possible reduction before one sets out to "SIMD-ize" a program.

4 An example application

The *firing squad synchronization problem* (FSSP) is a classical problem of cellular automata theory (for more details see [14]): Determine a one-dimensional cellular automaton with von Neumann-neighborhood (communication with immediate neighbors only) and the following properties:

- The initial configurations of interest have the form $\mathtt{GZ}_0^{s-1}$ where $\mathtt{G}$ is called the *general state*, $\mathtt{Z}_0$ is the *quiescent state* and s is the size of the cellular array.

- The (local) *transition table* σ has the property that $\sigma(\mathtt{Z}_0, \mathtt{Z}_0, \mathtt{Z}_0) = \sigma(\mathtt{Z}_0, \mathtt{Z}_0, \mathtt{Border}) = \mathtt{Z}_0$ i.e. neither areas of quiescent cells nor the cell at the right border are able to initiate any activity by themselves.
- The cellular array eventually reaches the configuration $\mathtt{F}^s$. $\mathtt{F}$ is called the *firing state.*
- No cell fires before all others fire.
- The automaton is allowed to use a fixed number of states while working for arbitrary sizes. This excludes trivial solutions using a counter in each cell.

The figurative interpretation is that a general (the leftmost cell) wants to make all his soldiers fire synchronously using neighborhood communication only. In [16] it has been shown that at least $2s - 2$ transitions are necessary to achieve synchronization and time optimal solutions have been developed in [16], [2], [5], and [9] employing 16, 8, 7, and 6 states respectively.

In order to answer one of the prominent remaining question, whether there are time optimal solutions with less states, an algorithm performing a depth-first search for transition tables has been developed:

- The root node is a transition table full of undefined entries.
- The main loop simulates the behavior of the automaton beginning with the smallest number of cells.
- When an undefined entry is encountered (a *choice point*), each possible result for this entry forms a new branch of the search tree.
- When the simulation violates the specification of the FSSP (too early or too late firing), backtracking is initiated.
- There is a special heuristics for pruning subtrees which cannot yield new information: During backtracking it is checked whether the choice point under consideration can influence the error that led to backtracking. This can be done using a special mode of simulation called *error simulation.*
- There are two special cases where pruning can be done without error simulation.

4.1 SIMD Implementation

The techniques described in the preceding sections have been applied to the FSSP search algorithm which would be prohibitively slow when naively translated into a data-parallel program since there are three interwoven asynchronous loops: Simulation, backtracking and error simulation. The implementation uses a 16384-processor MasPar MP-1 and the data-parallel ANSI-C extension MPL. All measurements were done using FSSP search with four states.

Starting point for parallelization is a nonrecursive, sequential C-implementation which can be naturally decomposed into seven elementary operations: Make a simple simulation step (advanceCol); advance simulation to the next time step (advanceTime); advance simulation to the next array size (advanceSize); push a choice point on the stack (makeChoicePoint); backtrack; make an error simulation step (simulateError), get the next choice at a choice point (getNextChoice). It can be argued that even the sequential algorithm became more manageable by implementing control as a state machine (using **GOTO**s!) because the usual "structured" programming approach tends to produce complicated nested loops with contrived exit conditions.

The SIMD implementation keeps the state of control in a register and therefore control overhead is negligible. The initial set of operations was transformed using the rules in Section 2.2: Tuning a rarely used table access and removing one of the special case heuristics at backtracking yield a 6 % and 17 % improvement respectively. advanceCol and simulateError have an expensive transition table access in common. These accesses were factored out using splitting and subsequently merged into an operation recomputeEntry yielding a 50 % improvement. Similarly, some common stack maintenance code in backtrack and makeChoicePoint was factored out into an operation cleanTos. Finally, Formula 5 indicated that advanceCol and advanceTime should now be tested for equally often, so they were "unsplitted" to one combined operation. Table 1 shows statistics for the final set of operations.

Table 1: Probability, cost and optimal (probabilistic) testing frequency for operations of test application.

Operation	p_i [%]	c_i [ticks]	f_i [%]
advanceCol	28.8	343	20.7
recomputeEntry	37.4	215	29.7
makeChoicePoint	3.5	256	8.3
backtrack	3.7	700	5.2
getNextChoice	2.3	352	5.8
cleanTos	8.0	399	10.1
simulateError	16.1	241	18.5
advanceSize	0.2	372	1.7

Table 2: Execution times versus number of PEs used, for test application

# PEs	T [s]
16384	12.7
8192	22.1
4096	40.7
2048	76.8
1024	148.4

Clever instruction ordering or duplication alone give limited speedup (less than 10 %) but applied together they yield a 63 % improvement as compared to testing for each operation once in random order. Instruction ordering was done manually. All in all, the optimizations for decreasing SIMD overhead make the program three times faster than the basic approach.

Together with an effective dynamic load balancing scheme for distributing subtrees (see [14, 11]) which achieves a processor utilization of more than 80 % and incurs a communication overhead of less than 15 %, the program achieves about 38 times the performance of a sequential implementation on a SPARC-2 workstation. Table 2 shows execution times for different machine sizes. It would be interesting to have figures about the remaining SIMD overhead but this is difficult since implicit globalor operations and overhead due to indirect addressing complicate the picture.

5 A special case: Interpreter loops

```
initialization
LOOP
    IF currentInstruction = $I_1$ THEN execute $I_1$
    ⋮
    IF currentInstruction = $I_n$ THEN execute $I_n$
    IF TRUE THEN
        save results
        fetch next instruction
        fetch operands
```

Figure 5: MIMD interpreter as a test loop.

There is quite a number of papers on emulating MIMD behavior (e.g. [3, 7, 8, 10]) which on the first glance are based on a slightly different road to solving the problem: The SIMD machine can interpret a locally stored program written in a RISC like machine language. However, Figure 5 shows that such an interpreter can be viewed as a special case of the general test loop of Figure 2. So far, we have not used this approach because it has a number of problems:

- In order to limit SIMD overhead there can only be a small number of simple instructions.
- Accessing instructions and operands requires several indirect memory accesses which are very slow on contemporary SIMD machines. Typically this takes one or two orders of magnitude more time than executing an instruction like add. Finite state control as in our FSSP example, in contrast, has almost no control overhead.
- Some of the operations of the FSSP might require hundreds of machine instructions which have to be interpreted one by one.
- It is not even clear whether a program as complex as the FSSP example would fit into the local memory of a MasPar PE. Complicated and time consuming swapping schemes might be required.

For all these reasons, a pure interpreter approach cannot be expected to yield practically useful performance on todays machines. On the other hand, interpreters have the conceptual appeal that they can handle arbitrarily complex programs with a fixed number of instruction, whereas the number of operations derived from the control flow of a program can in principle grow without bound. For some applications it might therefore be a good idea to take the best out of both worlds: A small general purpose instruction set for flexibility, and additional coarse-grained instructions specifically tuned for the program to be executed which do most of the real computation.

We now apply the techniques derived in the preceding sections to the interpreter approach. This can also serve as an example how these techniques can be adapted to incorporate other kinds of additional knowledge about control flow.

5.1 Modeling interpreters by Markov chains

In [1] interpreters are modeled using a Markov chain by assuming that instructions are independent and that an instruction I_i can be fully characterized by its probability of occurrence p_i and its cost c_i. This can be viewed as a special case of our Markov model for arbitrary test loops from Section 3. We introduce one operation for each instruction plus one special operation o_0 for accessing instructions and operands (with cost c_0). We know the control flow of the interpreter. An instruction is always followed by o_0 and o_0 is followed by one of the instructions according to their probabilities. All other transitions are impossible.

$$p_{ij} = \begin{cases} 1 & : \quad i = 0 \text{ and } j > 0 \\ p_i & : \quad i > 0 \text{ and } j = 0 \\ 0 & : \quad \text{all other cases} \end{cases}$$

As in Section 2.1, performance can be increased by optimizing the test loop. The special structure of control flow implies that every sensible test loop can be written as $S_1;o_0;S_2;o_0;\ldots;S_k;o_0$ (up to cyclic permutation). Where each S_j is a nonempty subset (called *subinterpreter* in [3]) of the instruction set. Its instructions can be tested for in some arbitrary order. Test loops of a different form would contain tests which can never be successful.

5.2 The NP-hardness of subinterpreter scheduling

The interpreter loops described in Section 5.1 have a considerably simpler structure than general test loops. So, we might hope that there are efficient methods for optimizing them. We now show that this is not the case. Consider the decision version of our optimization problem:
SUBINTERPRETER SCHEDULING
INSTANCE: Instructions $I_1, \ldots, I_n$ and costs $\{c_0, c_1, \ldots, c_n\} \subseteq \mathbf{N}$; probabilities $p_i \in \mathbf{Q}^+$ ($\sum_{i=1}^n p_i = 1$), a subinterpreter count $k \in \mathbf{N}$, a test loop length m and a cost bound $\overline{C} \in \mathbf{Q}^+$.
QUESTION: Is there a test loop with k subinterpreters and $k + \sum_{j=1}^k |S_j| = m$ for which the expected cost per executed instruction is $C \leq \overline{C}$?

Theorem 1 *SUBINTERPRETER SCHEDULING is* **NP**-*hard.*

Proof: Consider the well known **NP**-complete partition problem (quoted from [4]):

PARTITION
INSTANCE: A finite set A and a "size" $s(a) \in \mathbf{Z}^+$ for each $a \in A$.
QUESTION: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \ ?$$

We now transform an instance of PARTITION into an instance of SUBINTERPRETER SCHEDULING:
Let $n = |A|$, $\{I_1, \dots, I_n\} = A$, $c_0 = 0$, $c_1 = \cdots = c_n = 1$, $p_i = s(I_i)/\sum_{a \in A} s(a)$, $k = 2$, $m = n + k$ and $C = \frac{3}{4}n$. This is a legitimate instance of the SUBINTERPRETER SCHEDULING problem and it can be constructed in polynomial time. Consider an optimal test loop for this instance. It must have the form $S_1;o_0;S_2;o_0$ with $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$. Let $\alpha = \sum_{I_i \in S_1} p_i$. It is sufficient to show that the cost measure C is $\frac{3}{4}n$ if $\alpha = \frac{1}{2}$ and larger in all other cases.

The test loop can be modeled using a Markov chain with the following four states:

A_1: About to interpret S_1 and control flow waits for an instruction from S_1.

A_2: About to interpret S_2 and control flow waits for an instruction from S_2.

W_1: About to interpret S_1 but control flow waits for an instruction from S_2.

W_2: About to interpret S_2 but control flow waits for an instruction from S_1.

When the Markov chain is in state A_1 we can next book the successful execution of an instruction and the test loop will next be ready to interpret an instruction from S_2. With probability $1 - \alpha$ the control flow will also wait for an instruction from S_2 now, resulting in a transition to state A_2. In a similar way the other transition probabilities can be found resulting in the following Markov chain:

$$\begin{tikzcd} W_2 \arrow[r, shift left, "1"] & A_1 \arrow[l, shift left, "\alpha"] \arrow[r, shift left, "1-\alpha"] & A_2 \arrow[l, shift left, "\alpha"] \arrow[r, shift left, "1-\alpha"] & W_1 \arrow[l, shift left, "1"] \end{tikzcd}$$

Solving the corresponding eigenvector equation (as in Equation 2) yields the equilibrium probability to find the Markov chain in either of the active states A_1 or A_2:

$$a = \frac{1}{2(\alpha^2 - \alpha + 1)}.$$

Each traversal of the test loop incurs a cost n and results in two state changes of the Markov chain. Therefore the expected cost per executed instruction is

$$C = \frac{n}{2a} = n(\alpha^2 - \alpha + 1). \tag{7}$$

This cost measure takes the minimum $\frac{3}{4}n$ for $\alpha = \frac{1}{2}$

□

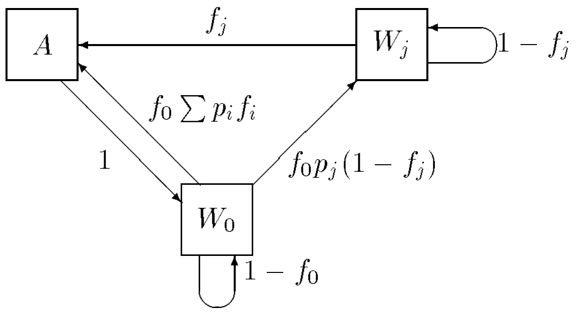

Figure 6: Markov model of probabilistic interpreter loop ($n+2$ states).

5.3 Probabilistic test loops revisited

As in Section 3.2 the combinatorial explosion involved in determining the optimal test loop can be avoided by looking at a probabilistic test loop. The derivation only has to be changed to take the special role of o_0 into account. Figure 6 shows the Markov chain model of the interpreter loop. The "wait for instruction" states W_j $(j > 0)$ have the same transition probabilities as the operations from Figure 4. What is new is that the active state has a mandatory transition into the state W_0. The Markov chain remains there until o_0 is encountered and subsequently makes a transition analogous to the transitions from A in Figure 4. A similar derivation as in Section 3.2 can now be used to derive the expected cost per instruction execution. The result is

$$C(f_0, f_1, \ldots, f_n) = \left(\frac{1}{f_0} + \sum_{i=1}^{n} \frac{p_i}{f_i}\right)\left(f_0 c_0 + \sum_{i=1}^{n} f_i c_i\right). \tag{8}$$

We see that by introducing the artificial "probability" $p_0 = 1$ we can write C as

$$C = \left(\sum_{i=0}^{n} \frac{p_i}{f_i}\right)\left(\sum_{i=0}^{n} f_i c_i\right) \tag{9}$$

which is completely analogous to Equation 4. Since the derivation of the optimal testing frequencies in Section 3.2 makes no use of the property $\sum p_i = 1$, we can immediately adopt the result

$$f_j = \frac{\sqrt{\frac{p_j}{c_j}}}{\sum_{i=0}^{n} \sqrt{\frac{p_i}{c_i}}} \text{ for } 0 \le j \le n. \tag{10}$$

Again, the probabilistic test loop is not directly useful as an implementation strategy but quantitative results can be a valuable tool. For example, the formula for the expected cost of the optimal test loop $C_{\text{opt}} = \left(\sum_{i=0}^{n} \sqrt{p_i c_i}\right)^2$ can be used to compare different instruction set designs without having to write an optimized interpreter for each of them. Once an instruction set has been decided upon, a heuristic procedure like the following might be used to construct a good interpreter loop:

1. Decide upon an approximate test loop length $m > n$. The larger m the better the potential speedup but the more difficult the fine tuning.

2. Use $k = \max(\text{round}(mf_0), 1)$ subinterpreters.

3. Make $\max(\min(\text{round}(mf_i), k), 1)$ replications of instruction i.

4. Partition the resulting multiset of instructions into k subinterpreters (sets of instructions) such that the summed probabilities for the individual subinterpreters are about equal. (Using some heuristics.)

5. Fine tune using hill climbing techniques or additional knowledge about the instruction set. It might also help to try different values for m.

6 Conclusions

There is no clear-cut border between SIMD algorithms and MIMD algorithms. A program with asynchronous control flow can be decomposed into a number of elementary operations which can emulate asynchronous behavior on a SIMD machine. For many applications, a small number of coarse-grained operations is sufficient resulting in an acceptable emulation overhead. However, more complicated programs may require a large number of operations or the decomposition into very fine-grained operations which resemble a machine instruction set. In this case, the overhead may become prohibitive.

Using the techniques developed here, it is possible to transform a program into a form more suitable for SIMD execution. The transformations can be applied manually but the most important ones are also sufficiently well defined in order to be performed by a compiler. Using a mixture of quantitative and qualitative tools, the emulation can be made considerably more efficient than a straightforward approach.

The most interesting quantitative tools used here are Markov chain models of control flow. They model the behavior of a test loop in a quite general setting and for probabilistic test loops it is even possible to derive closed form expressions for optimal testing frequencies. They also play a key role in proving that finding optimal deterministic test loops is an **NP**-hard problem.

Acknowledgements

I would like to thank S. Egner, M. U. Mock, R. Vollmar, P. A. Wilsey and T. Worsch for the many interesting discussions which helped to develop and refine the ideas described here.

References

[1] N. B. Abu-Ghazaleh, P. A. Wilsey, X. Fan and D. A. Hensgen, Variable Instruction Issue for Efficient MIMD Interpretation on SIMD Machines, Proc. 8th International Parallel Processing Symposium, (1994) 304–310.

[2] R. Balzer, An 8-state minimal time solution to the firing squad synchronization problem, Information and Control 10 (1) (1967) 22–42.

[3] H. G. Dietz and W. E. Cohen, A massively parallel MIMD implemented by SIMD hardware, Tech. Report TR-EE 92-4, Purdue University, 1992.

[4] M. R. Garey and D. S. Johnson, Computers and Intractability (Freeman and Company, New York, 1979.)

[5] H.D. Gerken, Über Synchronisationsprobleme bei Zellularautomaten, Masters thesis, University of Braunschweig, Germany, April 1987.

[6] R. Hanxleden and K. Kennedy, Relaxing SIMD control flow constraints using loop transformations, SIGPLAN Notices 27 (7) (1992) 188–199.

[7] D.Y. Hollinden, D. A. Hensgen and P. A. Wilsey, Experiences implementing the Mintabs system on a MasPar MP-1, Proc. 3rd Symposium on Distributed and Multiprocessor Systems (1992) 43–58.

[8] M. S. Littmann and C.D. Metcalf, An exploration of asynchronous data-parallelism, Tech. Report #684, Dept. of Computer Science, Yale University, 1988.

[9] J. Mazoyer, A six-state minimal time solution to the firing squad synchronization problem, Theoretical Computer Science 50 (1987) 183-238.

[10] M. Nilsson and H. Tanaka, MIMD execution by SIMD computers, Journal of Information Processing 13 (1) (1990) 58–61.

[11] C. Powley, C. Ferguson and R. Korf, Depth-first heuristic search on a SIMD machine, Artificial Intelligence 60 (1993) 199–242.

[12] P. Sanders, Suchalgorithmen auf SIMD-Rechnern – Weitere Ergebnisse zu Polyautomaten, Masters thesis, University of Karlsruhe, Germany, August 1993.

[13] P. Sanders, Emulating MIMD behavior on SIMD machines, Proc. International Conference Massively Parallel Processing Applications and Development (Elsevier), Delft, Netherlands (1994) (to appear).

[14] P. Sanders, Massively parallel search for transition-tables of polyautomata, Proc. 6th Workshop on Parallel Processing by Cellular Automata and Arrays, Potsdam, Germany (1994) (to appear).

[15] S. Tomboulian and M. Pappas, Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures, Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation (1990).

[16] A. Waksman, An optimum solution to the firing squad synchronization problem, Information and Control 9 (1966) 66–78.