Emulating MIMD Behavior on SIMD Machines

Peter Sanders
Informatik für Ingenieure und Naturwissenschaftler
Universität Karlsruhe
D-76128 Karlsruhe, Germany

Keywords: Average; Asynchronous control-flow; Markov chain; optimization; parallel search.

1 INTRODUCTION

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 *SIMD overhead* can become particularly large if the number of iterations through a loop varies from PE (processing element) to PE. A simple example are the number of iterations necessary to compute a point of the Mandelbrot set [1]. In [2] nested loops typical for numerical applications are discussed. A very general (but often inefficient) solution is to write the program in a RISC-like machine language which can be stored and interpreted locally [3, 4].

```
Let $s$ be a stack containing 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
        top($s$) := nextSibling(top($s$))
    ELSE push(firstSuccessor(top($s$)))
```

Figure 1: Nonrecursive generic depth-first search.

Complex instances of inherently asynchronous problems can be observed in nonnumeric applications. Throughout this paper depth-first tree search will be used as an example; Figure 1 gives the kernel of a nonrecursive, parallel algorithm searching for leaves which constitute a solution. A load balancer must ensure that each PE gets a different subtree of the search space represented by the subtree's root. Interior nodes are expanded by pushing their first successor on the stack. For leaf nodes, it is checked if they constitute a solution. Then the program backtracks to the next node with unsearched siblings.

On the average, the while-loop which performs backtracking will perform very few iterations; but on some PEs the number of iterations may be the full depth of the tree. Additional complications may be introduced by heuristics, or by loops inside the application dependent functions isLeaf, firstSuccessor, etc.

In Section 2, a simple and general approach for decomposing such an asynchronous program into synchronous operations is presented. Sections 3 through 5 discuss how this can be done efficiently. After Section 6 reports about the performance of a test application, Section 7 sums up the results.

2 THE BASIC APPROACH

The general idea for "SIMD-izing" an algorithm is very simple. It turns out that every MIMD program can be transformed into a SIMD program of the general form given in Figure 2. The o_i are *elementary operations* which do

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

Figure 2: Test loop executable by a SIMD machine.

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* of operation o_i. More generally, for many applications it makes sense to decompose the program (possibly dynamically) into a sequence of several test loops — each with a different set of tests — but, since the loops can be investigated one at a time, we can restrict ourselves to one loop.

A simple proof that a single unnested loop is always sufficient, can be taken from [3, 4] by observing that an interpreter for a machine instruction set has this form. By using a locally stored "machine program" and program counter, each PE can have its own flow of control. Due to its large interpretation overhead, this approach has not raised very much interest yet. However, the overhead can be dramatically reduced by tailoring the instruction set and even the interpreter itself to the one specific algorithm to be executed.

Since this idea is nonconstructive i.e., it is not obvious how to find an efficient instruction set and interpreter, it is useful to start with a different view. 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.

Again, every program can be cast into this shape: Eliminate recursion; inline procedures which contain loops, and implement the loop control by *goto*-statements. Then the control-flow graph can be partioned into cycle-free parts which yield one state of the automaton each. A jump goto label can then be replaced by the assignment state := label. This transformation

```
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.

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. It turned out that for some complex heuristic search algorithms this may even yield more readable algorithms than the usual structured programming approach which tends to produce complicated nested loops with contrived exit conditions.

3 LOOP UNROLLING

The main source of overhead in executing the test loops introduced in Section 2 on a SIMD machine is that all PEs for which g_i is false are idle during an execution of o_i. One important idea for reducing this overhead is to test cheap and frequently used operations more often than expensive and rarely used operations. As pointed out in [3, 4] this idea can be useless if the execution time is dominated by PEs where the distribution of needed operations deviates from the average case. But this argumentation neglects that a dynamic load balancer can relieve the load on these PEs.

For a simple example, the test loop for Algorithm 3 could be unrolled once such that each operation is tested twice. Then, one of the tests for the operation Solution could be removed if there are only very few solutions. Since a call to printSolution might require an expensive host communication, this simple measure could nearly double the performance of the algorithm. A problem with this idea is that is seems to involve a lot of trial and error. Therefore we want to model the situation mathematically.

Let c_i be the cost (or time) for testing operation o_i. (It can for example be measured using a profiler.) Furthermore, let p_i be the frequency of o_i in a typical dynamic trace of the program. (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 optimal throughput. This can be modelled using the probabilistic test loop

LOOP choose i with probability f_i; **IF** g_i **THEN** o_i **ENDIF 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* step. Stated probabilistically, this means the ratio between the expected time for doing a test $\sum f_i c_i$, and the probability a that the next test will perform productive work (i.e. it is currently needed in the asynchronous control-flow).

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

In order to determine a, the development of the state of execution of a PE is modelled using a Markov chain with the states A for "active" and W_j for "waiting for a test of o_j". Figure 4 shows the states and the transition probabilities. If a PE is waiting for a test of o_j, this operation is tested next with a probability of f_j and the Markov chain will make a transition to the active state, else waiting will continue. If the 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 next, there will be 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.

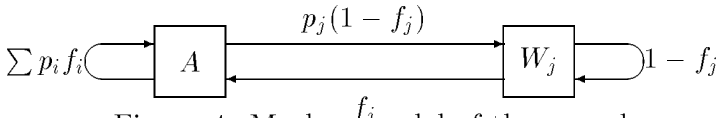

Figure 4: Markov model of the asynchronous control-flow.

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

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

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 first 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

$$a \sum \frac{p_i(1-f_i)}{f_i} = 1 - a \text{ or } a = \frac{1}{\sum_{i=1}^{n} \frac{p_i}{f_i}} \tag{2}$$

Substituting this result into equation 1 completes the cost function.

$$C(f_1, \ldots, f_n) = \left(\sum_{i=1}^{n} \frac{p_i}{f_i}\right) \left(\sum_{i=1}^{n} f_i c_i\right) \tag{3}$$

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 parts 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:

$$f_j = \frac{\sqrt{\frac{p_j}{c_j}}}{\sum_{i=1}^{n} \sqrt{\frac{p_i}{c_i}}} \tag{4}$$

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 loop unrolling can be estimated by comparing the resulting optimal cost with the straightforward case $f_i = 1/n$:

$$S = \frac{C_{\text{naive}}}{C_{\text{opt}}} = \frac{\sum_{i=1}^{n} c_i}{\left(\sum_{i=1}^{n} \sqrt{p_i c_i}\right)^2} \tag{5}$$

The value for C_{naive} also indicates that it is not a good idea to use a probabilistic test loop for the actual implementation since an equally naive deterministic test loop would only involve a cost around $1/2 \sum_{i=1}^{n} c_i$. Nevertheless, we can expect that the frequencies computed for the probabilistic case are also useful for a deterministic implementation.

4 OPERATION ORDERING

Another important issue is, how the tests should be ordered in order to make it as likely as possible that the next operation needed in the asynchronous control-flow is tested soon. For example, if the trees to be traversed by Algorithm 3 have a large branching factor, the asynchronous control-flow will be dominated by subsequences of the form Search; GetNextChoice; Search; GetNextChoice; ... and it is a good idea to match this structure in the unrolled loop.

The asynchronous control-flow can be described (at least approximately) using a Markov model where every operation corresponds to a state of the

Markov chain and p_{ij} gives the transition probability i.e. the probability that the operation o_i follows o_j in a typical dynamic trace.

Using this model, it is possible to predict the performance of a candidate test loop; e.g. 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 a test on this vector. By iterating a few times through the test loop, it is possible to approximate a cost function analogous to Section 3. (This turns out to be equivalent to modelling asynchronous control-flow *and* test loop by a Markov chain.)

However, this approach does not directly answer the question which arrangement of the test loop is optimal. Trying all possibilities leads to a combinatoric explosion. Nevertheless, it is probably possible to arrive at good solutions using hill climbing or other heuristics. In [3] a similar approach works reasonably well but no loop unrolling is considered which makes the problem simpler, and it also seems to severely limit the achievable improvement for nontrivial problems.

5 OPERATION SELECTION

In addition to dissolving loops as described in Section 2 there are a number of other important transformations on operations.

Splitting: A branch of a conditional statement which is rarely taken or very expensive can be made an operation of its own. For example this is the reason, why Solution is an independent operation in Algorithm 3. Sometimes the inverse operation of incorporating an operation into another is also useful in order to decrease control overhead. The formulae from Section 3 give a quantitative tool for assessing the impact of splitting on performance.

Simplification: In asynchronous programming languages most code need not be fine-tuned since only the inner loop is critical. In our approach however, the entire test loop is the inner loop. So, tuning rarely used operations can have an unexpected impact on performance. This effect is only mitigated using the optimizations from Section 3. On the other hand, asynchronous code often profits from optimized treatment of a number of special cases. In a SIMD program however, this approach may backfire since the code for the optimizations incurs additional SIMD overhead. Therefore, *removing* optimizations is sometimes the better optimization.

Merging: If two operations are almost identical like
o_1:α; state := o'_1
o_2:α; state := o'_2
they can be merged into the single operation
o_{12}: α; state = follow
if the calling operations assign the proper value to follow. 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.

6 PERFORMANCE RESULTS

The techniques described in the preceding sections have been applied to a heuristic search problem raised by an open question in cellular automata theory [5]. The implementation uses a 16384-processor MasPar MP-1 and the data-parallel ANSI-C extension MPL. Starting point for parallelization is a nonrecursive, sequential C-implementation which can be naturally decomposed into seven elementary operations.

Since these operations are quite coarse-grained and since state can be held in a register, control overhead is negligible. If it had been tried to interpret a program as in [3], control overhead would have been overwhelming since operations would be more fine-grained and each instruction fetch would require an indirect memory access which is very expensive on the MasPar.

Tuning a rarely used operation and removing a heuristic for a special case yield a 6 % and 17 % improvement respectively. Splitting is used twice and inverse splitting once. Together with a subsequent opportunity for merging, this results in a 50 % improvement. Clever instruction ordering or loop unrolling alone give limited speedup (less than 10 %) but applied together they yield a 63 % improvement as compared to testing 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 [5, 6]) 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 II workstation. Tables 1 and 2 show performance data for the final operation set and load balancing strategy. 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.

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

7 DISCUSSION AND 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 such that the emulation overhead is acceptable. 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, emulation overhead may become prohibitive.

Using the techniques developed here, it is possible to transform an asynchronous program into a SIMD program with the same semantics. The transformations can be feasibly applied manually and they are also sufficiently well defined in order to be performed by a compiler (at least partially). Using a mixture of quantitative and qualitative tools, the emulation can be considerably optimized.

From a theoretical point of view it is interesting that a simple but nontrivial rule for unrolling could be derived: The testing frequency of an elementary operation should be proportional to the square root of the ratio between its importance and its cost. As opposed to the testing frequency which can be modelled using a continuous model, optimizing the order of tests is a combinatorial optimization problem which appears to be harder to solve optimally.

References

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

[2] R. Hanxleden, K. Kennedy. *Relaxing SIMD control flow constraints using loop transformations.* SIGPLAN Notices, 188–199, 1992.

[3] M. Nilsson, H. Tanaka. *MIMD execution by SIMD computers.* Journal of Information Processing 13-1, 58–61, 1990.

[4] D.Y. Hollinden et al. *Experiences implementing the Mintabs system on a MasPar MP-1.* 3rd Symposium on Distributed and Multiprocessor Systems, 43–58, 1992.

[5] P. Sanders. *Massively parallel search for transition-tables of polyautomata.* PARCELLA 94, Akademie Verlag Berlin, 1994 (to appear).

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