Invited Paper

MIMD Execution by SIMD Computers

MARTIN NILSSON* and HIDEHIKO TANAKA**

SIMD computers cannot directly execute an MIMD language, but can interpret it by an SIMD interpreter. Such an interpreter circumvents the SIMD restriction of only allowing a single instruction stream, by taking the MIMD processes as its data (the MD of SIMD), while the interpreter itself is a single instruction stream. The speed of an SIMD interpreter for MIMD programs will depend on the organization of interpreter loop, and the optimal organization will depend on the MIMD program to be executed. We present a Markov chain mathematical model of primitive instruction execution in the interpreter loop. We describe an adaptive algorithm, based on this model, for dynamically optimizing the ordering of primitive operations of the interpreter loop. We also verify the adequacy of the model by experiments on a Flat GHC implementation.

1. Introduction

Parallel programming languages can be divided into MIMD languages and SIMD languages. SIMD languages are languages which allow only a single sequence of instructions, although each instruction may operate on vectors of data. Examples of such languages are Fortran 8x [1], with its array extensions, and *Lisp [5]. MIMD languages, on the other hand, allow multiple simultaneously executing processes, with independent control flows, such as GHC [6, 2].

MIMD languages are easier to use than SIMD languages, but SIMD computers are easier to build than MIMD computers. One way out of this dilemma is to execute MIMD programs on an SIMD computer by interpretation: The SIMD interpreter executes a series of instructions, and every process waits for its next instruction to be executed. The interpreter could be expressed as:

```
loop {
   for all processes waiting for operation A, do A;
   for all processes waiting for operation B, do B;
   for all processes waiting for operation C, do C;
   .
   .
   .
}
```

This code will efficiently execute processes which have their instructions in this order, e.g. A, B, C, or A, C. However, if a process would like to execute instructions in the order C, B, A, the relative performance will not be as good.

*Swedish Institute of Computer Science, Box 1263, S-164 28 KISTA, Sweden
**Department of Electrical Engineering, The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, 113 Tokyo

In this paper, we will analyse the effects of ordering the instructions in the interpreter loop on worst case and average performance. We introduce a Markov chain model for instruction execution, and based on this model, we will present an adaptive method for dynamically optimizing instruction ordering. We will also give some results for the algorithm applied to an experimental implementation of Flat GHC.

2. The Worst Case

As the goal of our optimization, we will try to minimise the number of instructions executed. We do not distinguish between fast and slow instructions; slow instructions can be viewed as several fast instructions always occurring in a sequence.

One of the first questions is: Can performance be improved if some instruction occurs more than once in the loop? If we want to minimise the worst case performance, the answer must be no, as we can see from the following "diagonal" argument: Consider the sequence A, B, C, . . . of instructions executed by this interpreter.

We will try to construct a process which runs as slowly as possible: As the process' first instruction, choose any instruction, X. As the process' next instruction, choose the instruction Y in the sequence A, B, C, . . which occurs as *far* after X as possible. Since there are N different instructions, this distance must be at least N, and greater than N if and only if there are more than one occurrence of some instruction between X and Y. Repeating this argument shows that there can be at most $N-1$ different instructions in the interpreter instruction stream between two instructions of the constructed process, for the process not to run slower than $1/N$ the speed of the interpreter.

The optimization problem is thereby reduced to finding a *permutation* of the interpreter instructions

such that the expected delay is minimised.

3. A Markov Chain Model of Instruction Execution

As a first approximation, let us assume that instructions are executed with independent probabilities p_i. The expected delay of an instruction in some process is the sum

$$D=\sum_{ij} p_i p_j d_{ij} \tag{1}$$

where d_{ij} is the distance between instruction i and j:

$$d_{ij}=\begin{cases} N & \text{if } i=j \\ j-i & \text{if } i<j \\ N-i+j & \text{if } i>j \end{cases} \tag{2}$$

Unfortunately,

$$\begin{aligned}\sum_{ij} p_i p_j d_{ij} &= \sum_i p_i^2 N+\sum_{i<j} p_i p_j (d_{ij}+d_{ji}) \\ &= \sum_i p_i^2 N+\sum_{i<j} p_i p_j N=N\sum_{ij} p_i p_j\end{aligned} \tag{3}$$

which is independent of the order between instructions.

A more meaningful model must take into account the correlation between instructions. We will use the following *Markov chain* model: Let q_{ij} be the probability that the next instruction is j, given that the current instruction is i, and assume that q_{ij} depends only on i and j. $Q=(q_{ij})$ then becomes the transition matrix for a Markov chain, for which the following three useful identities should hold:

$$\sum_j q_{ij}=1, \tag{4}$$

which says that every instruction is followed by some other instruction. This is true in our case, if we think of a process terminating instruction as followed by the first instruction of some other, new process.

$$\sum_i p_i=1, \tag{5}$$

which says that instructions 1 to N comprises the full instruction set.

$$\sum_i p_i q_{ij}=p_j, \tag{6}$$

which says that the probability that the next instruction is j, equals the sum of probabilities of instructions i, times their respective transition probabilities q_{ij}.

A useful property is that the vector of instruction probabilities $P=(p_i)$ is an eigenvector of Q, $QP=P$. Another useful property of ergodic Markov chains is that the rows of the matrix Q^k converges to P as k approaches infinity.

The average delay can now be expressed

$$D=\sum_{ij} p_i q_{ij} d_{ij}. \tag{7}$$

How does this ordering compare with exactly the *reverse* order? Let D_r be the delay of the reverse ordering. Then

$$\begin{aligned}D+D_r &= \sum_{ij} p_i q_{ij}(d_{ij}+d_{ji}) \\ &= \sum_{ij} p_i q_{ij} N+\sum_i p_i q_{ii} N=N+N\sum_i p_i q_{ii}\end{aligned} \tag{8}$$

which is independent of the permutation of the instructions.

This is a very useful result: First of all it tells us that if a certain ordering produces the best expected delay, then the reverse ordering produces the worst. The delays of an ordering and its reverse ordering are symmetric around

$$\frac{N}{2}\left(1+\sum_i p_i q_{ii}\right) \tag{9}$$

Since we showed in section 2 that the worst case cannot be worse than N, the best case cannot be better than

$$D_{\text{best}} \le N\left(1+\sum_i p_i q_{ii}\right)-N=N\sum_i p_i q_{ii} \tag{10}$$

The sum expresses the probability that an instruction is followed by itself. We can see that if this value approaches one, the delay worsens quickly towards the worst case, N.

We can illustrate this result through an example of a sequence where instructions are followed by themselves with high probabilities: The sequence A, A, A, A, B, B, B, B cannot be executed in less than seven cycles, regardless of the ordering of interpreter instructions. If this sequence is changed into A, B, A, B, A, B, A, B, it can be executed in four interpreter cycles.

Let us now see what happens if we switch the order of two adjacent instructions a, b in the interpreter. The expected delay is

$$\begin{aligned}D&=\sum_{ij} p_i q_{ij} d_{ij} \\ &= \sum_{i\neq a,b} p_i q_{ia} d_{ia}+\sum_{i\neq a,b} p_i q_{ib} d_{ib} \\ &\quad +\sum_{j\neq a,b} p_a q_{aj} d_{aj}+\sum_{j\neq a,b} p_b q_{bj} d_{bj} \\ &\quad +p_a q_{aa} d_{aa}+p_b q_{bb} d_{bb} \\ &\quad +p_a q_{ab} d_{ab}+p_b q_{ba} d_{ba}+\sum_{i,j\neq a,b} p_i q_{ij} d_{ij}\end{aligned} \tag{11}$$

The expected delay for the interpreter with instructions a and b interchanged is

$$D'=\sum_{ij} p_i q_{ij} d'_{ij} \tag{12}$$

where, for $i, j\neq a, b$,

$$d'_{ia}=d_{ib},\ d'_{ib}=d_{ia},\ d'_{aj}=d_{aj},\ d'_{bj}=d_{aj} \tag{13}$$

and

$$d_{ii}'=d_{ii}=N \tag{14}$$

so

$$D-D'=\sum_{ij} p_i q_{ij}(d_{ij}-d'_{ij})$$

$$= \sum_{i \neq a,b} p_i(q_{ia}(d_{ia}-d'_{ia})+q_{ib}(d_{ib}-d'_{ib})) + \sum_{j \neq a,b} p_a q_{aj}(d_{aj}-d'_{aj}) + \sum_{j \neq a,b} p_b q_{bj}(d_{bj}-d'_{bj}) + p_a q_{ab}(d_{ab}-d'_{ab}) + p_b q_{ba}(d_{ba}-d'_{ba}) \quad (15)$$

Using the fact that

$$d_{ia}-d'_{ia}=d_{ia}-d_{ib}=-1 \quad (16)$$

and, similarly,

$$d_{ib}-d'_{ib}=1,\ d_{ib}-d'_{ib}=1,\ d_{bj}-d'_{bj}=-1 \quad (17)$$

equation (15) becomes

$$D-D'=\sum_{i \neq a,b} p_i(q_{ib}-q_{ia}) + \sum_{j \neq a,b} (p_a q_{aj}-p_b q_{bj}) + p_a q_{ab}(2-N)+p_b q_{ba}(N-2) \quad (18)$$

By the identities (4)-(6) for p_i, and q_{ij}, (18) can be simplified to

$$D-D'=N(p_b q_{ba}-p_a q_{ab}) \quad (19)$$

This expression depends solely on the difference between the probabilities of instruction a following instruction b, and instruction b following a. This important property is the theoretical basis of our algorithm.

4. An Adaptive Ordering Algorithm

Equation (19) is so important because it enables us to devise a very simple rule for improving the instruction ordering: *We can switch adjacent instructions if the corresponding probability difference becomes negative, so that the expected delay monotonically decreases.*

It is easy to show that this method will always converge: If we cannot find any two elements to switch, we obviously already have a minimum. If we can find two such elements, the expected delay will decrease. Since there are only a finite number of permutations, we must reach a minimum within this number of exchanges, since no permutation will be repeated during the search. Note that any permutation can be produced by a number of exchanges of adjacent elements.

Given p_i and q_{ij}, an algorithm for optimizing the instruction ordering is the following:

Algorithm 1—Static Optimisation

- Start with a random permutation. Compare the expected delay with the reverse order permutation, and select the permutation with minimum expected delay.
- Iterate through all elements, comparing adjacent elements, and exchanging them if this lowers the expected delay.

The first step of the algorithm ensures that the expected delay is always better than $(D_{\text{worst}}+D_{\text{best}})/2$.

The assumption that p_i and q_{ij} are known in advance is excessively strong. These probabilities will in any case vary as the executed program changes. However, we can estimate probabilities dynamically as we execute the program: If we keep a table f_{ij} which contains the number of transitions from instruction i to j, and is updated on every instruction execution the expected value of f_{ij}, after n instructions, will be

$$f_{ij}=n\,p_i q_{ij} \quad (20)$$

Thus, the expected value of the difference

$$f_{ij}-f_{ji} \quad (21)$$

is proportional to $D-D'$, and immediately tells us whether instructions i and j should be exchanged.

Thanks to this simple form of the ordering test, it can be implemented costing the interpreter very little overhead.

Unfortunately, we cannot keep the first step of algorithm 1, since it is impossible to find out which is the better of an ordering and its reverse, without frequency data. However, according to our experience, the algorithm normally converges quickly to a good ordering, regardless of the initial ordering, so it seems that the first step is not so important.

For an adaptive version of the algorithm, we should reset the frequency table to zero at regular intervals, so as to make sure that changes of the program are reflected more quickly in the frequency table.

There is one important problem we have not dealt with: What if $f_{ij}-f_{ji}=0$? This happens often initially in the algorithm, when f is zero everywhere, and can trap us into a cycle of local minima far from the optimal ordering: If we *exchange* instructions i and j, it can happen that we will only exchange one particular instruction with all the other instructions in turn. If we *don't exchange*, it can happen that no instructions are ever exchanged.

In order to avoid such problems, we exchange i and j with probability 1/2, if $f_{ij}-f_{ji}=0$. This rule turns out to work very well in practice. Our final version of the algorithm is the following:

Algorithm 2—Adaptive Optimization

- Start with a random permutation. Keep a two-dimensional table of frequences of instruction transitions, f_{ij}, initially zero.
- During execution, increment f_{ij} for every i to j transition.
- After an instruction in the interpreter loop has been executed, check whether that instruction should be exchanged with the instruction preceding it, by checking the sign of $f_{ij}-f_{ji}$. If this difference is zero, exchange the elements with probability 1/2.

5. Results and Discussion

We have applied the adaptive algorithm on a small pseudo-parallel SIMD interpreter for Flat GHC, ex-

ecuted on a conventional computer. We first compile Flat GHC programs down in stages, in the same way as in [3, 4], into a low-level intermediate language consisting of 14 different instructions, which is sufficient for executing all of Flat GHC and allows fairly accurate testing of the algorithm.

For two traditional benchmarks, concatenate and 4-queens, we measured a distinct speed-up by adaptive reordering: The worst and best number of interpreter cycles for concatenate differed by a factor of about 2, while the number of cycles for 4-queens differed by about 20%. Instruction orderings seem to converge rapidly towards a good ordering, even if the initial ordering is not very good.

We have not solved the problem of the ordering getting stuck in a local minimum. So far our tests indicate that this is not a serious problem. However, it may be a good idea to stop execution at regular intervals and reshuffle the instructions.

6. Conclusions

Assuming best worst case, and a Markov chain model of instruction execution, we have deduced some interesting properties of SIMD interpreters for MIMD languages:

- The expected instruction delay D for an ordering, and for its reverse, D_r, satisfy $D+D_r=N+N\sum_i p_i q_{ii}$ and $N\sum_i p_i q_{ii} \leq D \leq N$.
- A good heuristic for reducing the delay is to exchange two adjacent elements a and b if the frequency difference of the transitions of a to b and b to a is negative.

Based on these results we have constructed an adaptive algorithm for optimizing the instruction order.

7. Acknowledgments

Discussions with members of the Special Interest Group of the Inference Engine at the university, and with members of the Parallel Programming Systems Working Group at ICOT have been helpful and stimulating. We are grateful to Takeshi Shimizu of Tokyo University, who was very helpful in preparing this manuscript.

This work was supported by the Japanese Ministry of Education, and the Swedish National Board for Technical Development.

References

1. American National Standards Institute: *American National Standard for Information Systems. Programming Language Fortran.* S8 (X3.9-198x). Revision of X3.9-1978. Draft S8, Version 99. New York, April (1986).
2. Furukawa, K. and Mizoguchi, F. (Eds.). *The Parallel Programming Language GHC and its Applications.* Kyoritsu Publishing Co. Tokyo, 1987. (In Japanese).
3. Nilsson, M. and Tanaka, H. *Massively Parallel Implementation of Flat GHC on the Connection Machine.* In Proc. Fifth Generation Computer Systems, ICOT, Tokyo, Japan. November, **3** (1988), 1031-1040.
4. Nilsson, M. *Parallel Logic Programming for SIMD Supercomputers and Massively Parallel Computers.* Doctorate Thesis in Information Engineering, University of Tokyo, Japan. March (1989).
5. *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Corporation, Technical Report 87-7. April (1987).
6. Ueda, K. *Guarded Horn Clauses.* Doctorate Thesis in Information Engineering, University of Tokyo, Japan. March (1986).

(Received June 2, 1989)