(2)

CPSY88-22

SIMDアーキテクチャと超並列論理プログラミング
SIMD Architecture and Superparallel Logic Programming

Martin Nilsson and Hidehiko Tanaka
田中英彦研究室電機工学科東京大学
Hidehiko Tanaka Lab., Department of Electrical Engineering,
University of Tokyo, Hongo 7-3-1, Bunkyo-ku, 113 Tokyo

あらまし SIMDアーキテクチャは、超並列論理プログラミングに大変適しているアーキテクチャということを示す。そして、どのようなSIMDアーキテクチャが良いかを検討する。そのために、アーキテクチャの種類が違うSIMDコンピュータを二つ比較する。一つは、日立のベクトル並列スーパーコンピュータS-820、もう一つは、Thinking Machines社のConnection Machineである。前者に対しては、処理系を作成し、実測した。後者に対しては、命令レベルの詳細なシミュレータでベンチマークを行った。

Abstract SIMD architectures are interesting for massively parallel execution of logic programming languages. In this paper, we first motivate this conjecture. Then, in order to see what kind of SIMD architecture is suitable, we compare implementations of Flat GHC for two computers, representatives of two very different SIMD architectures: One is Hitachi's vector parallel supercomputer, S-820. The other is Thinking Machines' Connection Machine. For S-820, we have measured a real implementation. For the Connection Machine, we use a detailed instruction-level simulator for measurement.

1 Introduction

Parallel logic programming languages, such as GHC [17], are attractive for efficient execution on parallel computers, because of their relative simplicity and expressive power.

For massively parallel computers, with thousands of parallel processes, it is especially important that languages are simple, so that overhead is minimized. In order to investigate exactly how efficiently logic programming languages can be implemented, and what the bottlenecks are, we have chosen to implement the language Flat GHC. We selected this language since it seemed to be one of the simplest of its kind, and since it has been shown to have reasonable expressional power, e.g. [2].

After converting Flat GHC programs into an even simpler, intermediate language called Fleng [9], we execute programs by an interpreter. Care has been taken so that this interpreter consists of only vector operations. The interpreter algorithm is essentially the same for both S-820 and Connection Machine implementations. (The fine details of Fleng, and the translation from Flat GHC are described in detail in different papers [9], [10], [12]. Such knowledge is not necessary for the understanding of this paper.)

We find that S-820 is much faster than the current Connection Machine for our implementation but that this is essentially because of much faster hardware. More importantly, the S-820 does not scale easily to larger degrees of parallelism, while the Connection Machine does, so the Connection Machine does seem promising for the future.

First we will give a quick review of GHC, Fleng, and vectorization of the Fleng interpreter.

1.1 Flat GHC and Fleng

A Flat GHC program looks just like a Prolog program, but contains an *commit* operator in every clause body. If we think of this operator as a Prolog "cut," the execution of Flat GHC becomes very similar Prolog, with the difference that several clauses may be executed in

parallel.

The goals in a clause body in front of the commit symbol are called guard goals. A Fleng clause is similar to a GHC clause, but there are no guard goals. Also, even if execution of a body goal fails, it will not affect any other goals being executed. These two points are very important for minimizing the implementation overhead of the interpreter.

Fortunately, Flat GHC can still be automatically translated into Fleng by a reasonably simple process. The conversion can be done essentially by partially evaluating unification as far as possible in the guards, and replacing calls to system predicates with slightly modified versions.

1.2 Vectorization

For efficient vectorization, or "SIMD"-fication, all code in the interpreter loop should be vectorizable. This makes it important to make interpreted programs homogeneous, and minimize the number of different operations.

This has some consequences such as that it seems that structure sharing is more suitable than structure copying, since copying is hard to vectorize. The penalty in the form of increased time for garbage collection is also expensive. The state of a process must be small, since we are considering very high degrees of parallelism (tens of thousands). It also seems advantageous to represent data structures by binary cells, rather than varisized records, to improve homogeneity of the implementation.

The general structure of our interpreter is not unlike that of a traditional Prolog interpreter, and even less different from that of a Flat GHC or Fleng interpreter [10]. The main difference is that all processes who want to do the same operation are collected, and their data are operated on as vectors, where the N-th elements are data belonging to process N.

1.3 Related work

Kanada [6],[7] uses a compiled approach for executing OR-parallel Prolog search programs on a supercomputer. We suggested implementing an AND-parallel committed-choice language as a tight loop vectorizable interpreter for a supercomputer [9], and for the Connection Machine [11]. Tatsuguchi [16] has used this approach to implement OR-parallel and restricted AND-parallel interpreters for a vector parallel supercomputer. Bawden and Agre [1] have implemented a non-communication version of Scheme on the Connection Machine. However, we don't know of any comparative study of implementation of the same language on both a pipelined supercomputer, and the Connection Machine.

1.4 Paper overview

In section 2, We will first describe the building blocks we are going to use for implementation. Then we demonstrate the convenience of the SIMD approach, by showing methods for mutual exclusion and distributed unification (variable binding) in section 3. The basic structure of the interpreter is outlined in section 4. In section 5, we show benchmark results of the two implementations, and finally, in section 6, we discuss the results.

2 Vector Operation Primitives

For SIMD computers, we cannot not use jump instructions or procedure calls. In order to execute operations conditionally for individual processors, we must allow all operations to be conditional on a flag or *mask* operand: If the mask is true, the operation is executed as usual, but if the operand is false, the operation becomes a no-op. For instance, a conditional instruction `move(s,d,m)` takes the three operands: source `s`, destination `d`, and mask `m`. This instruction would for processor `i` move the contents of memory location `s[i]` to location `d[i]`, iff the flag `m[i]` is set to true.

Such an operation exists in the machine language of S-820. It is automatically generated by the compiler, from a Fortran program such as:

```
      DO 10 I = 1,N
        IF (M(I)) D(I) = S(I)
10    CONTINUE
```

Here, data is only moved locally inside a process, not between different processes, if we think of the collection of the i-th vector positions as a process.

We would like to have the following similar "traditional" instructions, with their obvious interpretations, as part of our SIMD instruction set:

`movs(s,d,n)`	(distribute a scalar constant to all processors),
`add(s1,s2,d,m),`	(arithmetic),
`sub(s1,s2,d,m),`	
`mul(s1,s2,d,m),`	

```
div(s1,s2,d,m)
and(s1,s2,d,m)       (bitwise boolean and),
or(s1,s2,d,m)        (bitwise boolean or),
xor(s1,s2,d,m)       (bitwise boolean xor),
movmask(ms,md,m)     (move mask),
cplmask(ms,md,m)     (complement mask)
```

These operations either already exist for the Connection Machine, or can be easily implemented [3], [4]. They are all in the machine instruction set for the S-820.

We also want to have a few instructions for comparison. The result of the comparison is stored in a mask vector:

```
cmpeq(s1,s2,md,m)    (equality),
cmpne(s1,s2,md,m)    (inequality),
cmplt(s1,s2,md,m)    (less than)
```

The following operations come in handy, although definable in terms of the previous operations:

```
adds(s1,s2,md,m)        (add scalar to vector),
cmpeqs(s1,s2,md,m)      (compare scalar with vector),
cmpnes(s1,s2,md,m)      (analogous),
ormask(ms1,ms2,md,m)    (inclusive or of masks)
```

We would also like to have instructions which enable processes to communicate with each other by writing to, and reading from each other's local memory. We introduce two new instructions for this purpose, `store(s,d,x,m)`, and `load(s,x,d,m)`.

In Fortran, `store` can be expressed as:

```
      DO 10 I = 1,N
        IF (M(I)) D(X(I)) = S(I)
10    CONTINUE
```

`load` can be defined as:

```
      DO 10 I = 1,N
        IF (M(I)) D(I) = S(X(I))
10    CONTINUE
```

The S-820 has so called *list vector* instructions, with which these can be vectorized.

For the Connection Machine, we assume here that `store` never tries to store two data in the same location, i.e. that all `x[i]` for which `m[i]` are true, are different. We assume the same thing for the `load` instruction, so that it will never try to read several times from the same processor. The reason why we restrict these operations is that these operations become very costly with the full generality.

When we need fully general `load` and `store` operations, we use the operations `load_1_to_many(s,x,d,m)` and `store_many_to_1(s,d,x,m)`. The speed of these pairs of instructions differ by a logarithmic order in the number of processors, for the Connection Machine. `store_many_to_1` introduces non-determinism by storing values in the destination by overwriting, without control of *which* of the values are written. This operation is useful for arbitration of parallel processes.

For S-820, the list vector operations do not distinguish these cases.

We also need to be able to count or enumerate all processors with their corresponding mask bits set. The `count(d,m)` operation counts the number of mask bits, and fills `d` with this value. `enumerate(d,m)` could be described in Fortran as:

```
      TMP = 0
      DO 10 I = 1,N
        IF (M(I)) THEN
          TMP = TMP + 1
          D(I) = TMP
        ENDIF
10    CONTINUE
```

On the S-820, such operations can be vectorized, although it is often a better idea to compress vectors first, and then enumerate them by looking at the vector index after compression.

Finally, we would like some instructions for tag extraction and testing:

```
TAG(s,d,m)         (extract tag),
ISVAR(s,md,m)      (check if variable),
ISCONS(s,md,m)     (check if cons)
```

These operations are all local. They can easily be defined in Fortran by using the other available instructions.

3 Why SIMD?

We will give two convincing examples, where an SIMD approach is considerably simpler than a MIMD approach. These examples are in fact relevant in many other parallel programming contexts than logic programming.

3.1 Mutual exclusion

Mutual exclusion of a number of processes which want to write some certain memory position, is implemented by letting processes write their index into the cell they request. Then they read back the cell's contents. If the value read back equals the index, permission is granted.

The code for mutual exclusion looks in principle like this:

```
      DO 10 I = 1,N
        CONTENTS(DEST(I)) = I
 10   CONTINUE

      DO 20 I = 1,N
        IF (EXCLUDE(DEST(I)) .EQ. I) THEN
          CONTENTS(DEST(I)) = VALUE(I)
        ENDIF
 20   CONTINUE
```

For the Connection Machine, the sequence would be:

```
store_many_to_1(i,contents,dest)
load_1_to_many(contents,dest,tmp)
cmpeq(tmp,i,mask)
  store(value,contents,dest)
```

3.2 Distributed unification

In distributed unification, we have the problem of binding a variable to another, with a directed binding. Unless we are careful, it might happen that several variables which are bound asynchronously are bound in a circle. This is not acceptable, but avoiding it is hard; if we try to lock both variables, we risk deadlock. Another way is to impose a permanent ordering on all variables, and make sure that variable bindings are bound in the direction specified by the ordering. This scheme can work, but leads to substantial overhead.

For an SIMD implementation, *we only need a temporary ordering during the binding of the variables.* After binding, the ordering may safely be forgotten. No cycles can be created by binding variables, assuming that they have been fully dereferenced before binding. (This is not obvious, but is perhaps most easily seen by observing that the fact that variables are bound in tree structures is an invariant.)

4 The Interpreter

For vectorization, as much code as possible in the interpreter loop should be vectorizable. This makes it important that interpreted programs are homogeneous, and that the number of different operations is minimized. This has some far-reaching consequences for the implementation: First, compilation to machine code is not possible in general, since compilation implies specialization, and makes execution inhomogeneous. On the other hand, an interpreter performs general operations, which can be applied to many elements at a time. Second, structure sharing is more suitable than structure copying, since for copying is hard to vectorize, and the overhead becomes quite expensive. The penalty in the form of increased time for garbage collection is also expensive. Third, the state of a process must be small, since we are considering very high degrees of parallelism (tens of thousands). For this reason, we cannot use linear stacks: With one stack area for every vector element, the memory requirements would be gigantic, even for a supercomputer. Necessary memory cells have to be allocated from a common memory pool, and kept together by pointers. Fourth, representing data structures by binary cells, rather than varisized records, improves homogeneity of the implementation.

The general structure of the interpreter is not unlike that of a traditional Prolog interpreter, and even less different from that of a Flat GHC or Fleng interpreter. The main difference is that all processes who want to do the same operation are collected, and their data are operated on as vectors, where the N-th elements are data belonging to process N.

The interpreter consists of three blocks: AND, OR, and UNIFY.

Each block takes a queue of processes as its input and produces a new queue of processes, ready to be executed by the next block.

The AND block takes trees of goal literals as input and pairs the goals with predicate definitions. It also adds a shared *Trust*-cell. It executes built-ins for arithmetic.

The OR block inputs pairs of goals and definitions, and produces pairs of goals and candidate clauses. It creates new environments, and "pre-commits" environments which are used for active unification.

The UNIFY block handles unification both (passive and active). It also handles dereferencing, variable binding, and commitment. For successful unifications, new goals are produced for the AND block.

5 Benchmark Results

We have implemented the basic interpretation algorithm as a C program, which is macro expanded into a Fortran program for the S-820. For the connection machine we have implemented an emulator of the instructions in C.

By running concatenate-type benchmarks through the interpreter, and counting the executed instructions, we found approximate values for the different classes of instructions, as shown in the table below. For the S-820, we used a system timer for computing the total scalar and vector CPU-time.

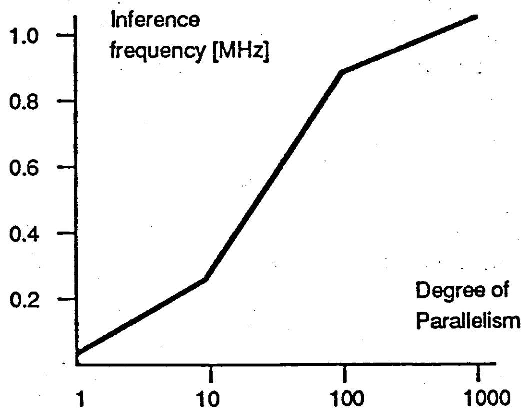
We considered `concatenate` suitable as a benchmark, since it is defined and executed in the same way in most logic programming languages.

Instruction group	Time [μs]
mask operations	1
`mov, add, etc`	12
`enumerate`	200
`count`	400
`mul`	380
`div`	380
`store`	450
`load`	900
`store_many_to_1`	15,000
`load_1_to_many`	15,000

5.1 The S-820

For a degree of parallelism of about 1000, the reduction frequency of the S-820 was approximately 500 kHz. Raising the degree of parallelism over this level, did not seem to affect the reduction frequency very much.

By optimizing the S-820 interpreter, we could speed up the interpreter by a factor of about two, to 1.1 MHz. By varying the degree of parallelism, we produced the following diagram of the reduction frequency as a function of degree of parallelism for the optimized version (note that the scale is lin-log):



5.2 The Connection Machine

Detailed instruction timing of the Connection Machine has not been published as far as we know, but by "reverse calculating" from the data spread through the articles [3], [4], [14], we have made the estimations as given in the following table:

Running the benchmark on the Connection Machine simulator produced the following total instruction times:

Instruction group	Frequency	Total time per process reduction [ms]
mask operations	341	0
`mov, add, etc`	625	8
`enumerate`	107	21
`count`	11	4
`mul`	5	2
`div`	5	2
`store`	318	143
`load`	136	122
`store_many_to_1`	11	165
`load_1_to_many`	130	1,950
Total	1,689	2,417

The peak process reduction frequency for a 4 MHz-clock, 65,536-processor Connection Machine will thus be approximately $65,536/2.417\,\text{Hz} \approx 27\,\text{kHz}$. For lower degrees of parallelism, the reduction frequency will be proportionally less.

6 Discussion and Conclusions

It is clear that S-820 is much faster than the current Connection Machine, mainly depending on the S-820's much faster hardware. The simulation data shows clearly that execution time for the Connection Machine is dominated by `load_1_to_many`. Since the "physical" parallelism of the S-820 is rather small (the number of pipelines times the number of pipeline stages), the `load_1_to_many` type contention is not as serious for S-

820 as it is for the Connection Machine. However, even if the time for `store_many_to_1` was reduced to that of `store`, and `load_1_to_many` to that of `load`, the speed would increase by a factor of just 5.7.

If the Connection Machine were built with the same state-of-art device technology as the supercomputer S-820, allowing the same clock frequency (250 MHz), the maximum process reduction frequency would rise to about 1.7 MHz.

These numbers suggest that for our implementation, a 65,536-processor Connection Machine would have a performance comparable to that of S-820, and sequential computers [15], if built using the same clock speed.

Our interpreter is certainly not optimal, so improving it will also improve the reduction frequency. But since the dominating factor for execution time depends on reading the user program, using the `load_1_to_many` instruction, which must be done by any kind of interpreter, so changes are not likely to radically alter our estimates.

Although it seems the current Connection Machine is slower than S-820 for this implementation, it is important to note that the Connection Machine is easily *scalable* to a larger number of processors, while the S-820 is not. Thus, it seems that a Connection Machine architecture has bright outlooks for the future.

7 Acknowledgments

This work was supported by the Japanese Ministry of Education, and the Swedish National Board for Technical Development. We have benefited very much from 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. We are especially grateful for vivid discussions with Kanada-san and Tatsuguchi-san.

References

[1] Bawden, A., Agre, P.E.: *What a parallel programming language has to let you say.* MIT AI Memo 796. September 1984.

[2] Furukawa,K. and Mizoguchi,F. (Eds.): The Parallel Programming Language GHC and its Applications. Kyoritsu publishing Co. Tokyo, 1987. (In Japanese).

[3] Hillis, W.D.: *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[4] Hillis, W.D. and Steele, G.L., Jr.: *Data Parallel Algorithms.* CACM, Vol. 29, No. 12. p 1170-1183.

[5] *HITAC S-810 Processor's Handbook.* Manual no. 6010-2-001. Hitachi, Ltd. September 1984. (In Japanese)

[6] Kanada, Y.: *High-speed Execution of Prolog on Supercomputers.* In Proc. 26th Programming Symp., Information Processing Society of Japan. 1985. p 47-55. (In Japanese)

[7] Kanada, Y.: *High-speed Execution of Prolog on Supercomputers - Realization and Performance of different models of OR-vector execution.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 12, July 1987. p 1-10. (In Japanese).

[8] Kawabe,S., Kobayashi,F., Murayama, H., et al.: *S-820 — 2 GFLOPS Peak Performance by a Single Processor.* Nikkei Electronics, No. 437. December 1978. p 111-125. (In Japanese)

[9] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, 1986. p 209-216. Proceedings also published as Springer LNCS 264.

[10] Nilsson, M. and Tanaka, H.: *The Art of Building a Parallel Logic Programming System.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, June, 1987. p 155-163. Proceedings also to appear as Springer LNCS.

[11] Nilsson, M. and Tanaka, H.: *A Proposal for implementing GHC on the Connection Machine.* In Proc. IEEE Region 10 Conf. p 821-825. Seoul, August, 1987.

[12] Nilsson, M. and Tanaka, H.: *Converting FGHC Clauses with Guards into Clauses without Guards.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 17, July 1988. (In bad Japanese).

[13] Nilsson, M. and Tanaka, H.: *A Flat GHC Implementation for Supercomputers.* To appear in Proc. Int. Conf. Symp. Logic Programming, Seattle, August 1988.

[14] Stanfill, C. and Kahle, B.: *Parallel Free-Text Search on the Connection Machine System.* CACM, Vol. 29, No. 12. p 1229-1239.

[15] Takami, S.: *Fujitsu announces integrated AI related products, AI support, and a 2 MLIPS Prolog Compiler.* In Nikkei Electronics, No. 427, August 1987. p 78-79. (In Japanese).

[16] Tatsuguchi,K. and Muraoka,Y.: *Parallel Logic Programming Interpreters on Supercomputers.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 14, December 1987. (In Japanese).

[17] Ueda, K.: *Guarded Horn Clauses.* D.Eng. Thesis, Information Engineering course, University of Tokyo, Japan. March 1986.