PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

Massively Parallel Implementation of Flat GHC on the Connection Machine

Martin Nilsson and Hidehiko Tanaka
Hidehiko Tanaka Lab., Dept. of Electrical Engineering,
The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN

Abstract: We have investigated implementation of Flat GHC on hypercube-based, massively parallel SIMD computers. Implementations running on a simulator of a Connection Machine show that the peak number of process reductions per second could reach up to on the order of 400 kHz for programs with very much parallelism. Although this is not faster than traditional sequential implementations, the advantage with the hypercube SIMD approach is that it is *scalable*: the peak reduction frequency increases quickly, $O(n/log^2\ n)$, with the number of processing elements n. Analysis shows that the limit on the reduction frequency stems mainly from simultaneous read accesses of shared data by different processing elements.

Although this paper uses Connection Machine data for timing estimation, several results should be qualitatively applicable for other SIMD computers. Likewise, fundamental limitations of this implementation are relevant for SIMD-style interpreters of other MIMD languages as well.

1 Introduction

1.1 Motivation

What are good architectures for parallel programming languages? What are good programming languages for parallel architectures? This paper tries to contribute to the future answers to these two important questions.

SIMD computers are attractive as candidates for massively parallel computers, i.e. computers containing more than thousands of parallel processing elements: For the hardware implementor, construction is greatly simplified, as compared with MIMD computers. For the implementor, programming problems such as synchronization become much easier. However, the user is unlikely to be satisfied by having to program in a restricting SIMD language. One way to solve this trade-off, is by interpreting the user's MIMD program by a SIMD interpreter. Here, the interpreter takes the user's program (the MI of MIMD) as data (MD of SIMD).

We are particularly interested in interpreting concurrent logic programming languages by SIMD interpreters, since this kind of language allows very compact expression of interpreters. This property is particularly important for SIMD implementations, since compactness improves processor utilization. Logic programming languages such as Flat GHC [29], manage to do quite complex operations with only unification, plus a very small set of primitive operations. We have decided to use Flat GHC as our MIMD language candidate, since this language has been shown to be a powerful programming language [3].

Of the massively parallel computers currently available, one of the most interesting is the Connection Machine [5, 6, 28]. This is a hypercube-structured computer, which allows fast communication between arbitrary processing elements. We have implemented two different interpreters which run on top of a machine instruction simulator for the CM-2 version of this machine: One interprets a high level intermediate language Fleng, which is similar to Flat GHC. The other interpreter interprets low level machine instructions, closely resemblant of the Warren Abstract Machine [31]. The simulator is detailed enough to allow execution timing and analysis.

The main purpose of this paper is to find some indications of how, and how efficiently, logic programming languages like Flat GHC could be implemented on massively parallel computers, such as the Connection Machine.

1.2 SIMD Programming Model

A processing element of an SIMD parallel computer is similar to a traditional processor. The difference is that the instruction flow does not come from the element itself, but from a host computer used as a sequencer. The sequencer broadcasts program instructions in a sequence which cannot be controlled directly by individual processing elements.

Thus, the SIMD instruction set may not contain any instructions which attempt to alter program flow, i.e. JUMP, CALL, or RETURN instructions. Such instructions are only meaningful for the host processor which distributes the SIMD instructions to the parallel back-end computer.

The operands of SIMD instructions can be seen as *vectors*, where each element i of an operand vector gives the actual operand for a processing element i.

Since we sometimes need conditional operations, we allow 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 local memory location `s` to location `d`, iff the flag `m[i]` is set to true.

We can express this in pseudo-formal notation as:

```
for all processors i do
  if (m[i]) then d[i] := s[i]
```

We would also like to have instructions which enable processors to communicate with each other by writing to, and reading from each other's memory. For this purpose we will introduce the instructions `store(s,d,x,m)`, and `load(s,x,d,m)`. In pseudo-formal notation, `store` can be defined as:

```
for all processors i do
  if (m[i]) then d[x[i]] := s[i]
```

`load` can be defined as:

```
for all processors i do
  if (m[i]) then d[i] := s[x[i]]
```

We will give the full set of instructions in section 6.

1.3 The Connection Machine

The Connection Machine consists of 65,536 processing elements connected in a hypercube structure. It is attached as a back-end processor to a conventional computer, which sequentially broadcasts instructions, such as `mov` in the previous section, to all processing elements, via a microcontroller.

Each processing element contains its own memory, and performs operations on this memory according to the instructions from the host computer. Except for operating on their own memories, processing elements also have the ability to send messages to other elements in the hypercube. All message transfers are serial. The memory of every processor is also connected to the host's bus, so that the host is able to access the memory of every processing element, although this has to be done one element at a time.

The operand vectors, or in other words, the arguments of the SIMD instructions, are called *pvars* - parallel variables. Element 0 of a pvar resides in processor 0, element 1 in processor 1, etc.

An important feature which distinguishes the Connection Machine from other massively parallel SIMD computers is the concept of *virtual processors*. If more processing elements are needed than are physically available, processing elements can be *multiplexed*. Of course, the speed is reduced, but this mechanism provides a kind of graceful degradation. N processors can simulate kN processors with $1/k$ the amount of memory per processor, in k times the execution time.

We should also mention here that we have tried to be as accurate as possible in the description of the Connection Machine. We believe that we have been sufficiently careful to ensure that our data are adequate, but if there are any mistakes, they are solely those of the authors. There is no commitment or guarantee by Thinking Machines, Inc. whatsoever that any Connection Machine will actually perform according to these estimations.

1.4 Approach

By definition, SIMD computers cannot directly execute MIMD programs, since all processors have to execute the same instruction at the same time. However, an SIMD computer could execute an interpreter, which first executes an instruction for all processors which are waiting for this particular instruction. Next, it executes another instruction, for some other set of waiting processors. Large degrees of parallelism are possible, but there are serious limitations: One is that there is much communication overhead, and another is that many processes may be idle waiting for their next instruction to be executed.

There are several alternatives for constructing an interpreter for MIMD programs: One important design decision is *on which level* the interpreter should interpret the program. The high-level approach interprets the MIMD language almost as is. The low-level approach first compiles the MIMD language to an instruction set much closer to the machine.

At first sight, it seems that the low-level approach will be much faster than the high-level approach, but this does not always necessarily have to be true: Low-level compilation will require a larger intermediate language instruction set, which indirectly slows down the interpreter.

We have implemented two interpreters, a high-level version and a low-level version.

For the high-level interpreter, we first translate Flat GHC into Fleng, a similar language whose interpreter is simpler to implement as a very compact loop. This interpreter is similar to a traditional Prolog interpreter, with the difference that it operates on vectors instead of scalar variables. An alternative way of viewing execution is as many interpreters executing synchronously.

The low-level version interprets a variant of the Warren Abstract Machine [31] we call μWAM, for *mu*-tilated WAM.[1] This interpreter is similar to a standard byte-code WAM interpreter.

Both of these interpreters are defined using a small set of SIMD instructions. Each instruction corresponds directly to one or a few Connection Machine instructions. This makes it comparatively easy to estimate the execution time without having a Connection Machine, through emulating these instructions by a simulator, and counting

[1] *καὶ σὺ τέκνου...*

the frequency of each type of instruction.

1.5 Results

The simulations show that for our implementations, a 65,536-processor Connection Machine has a peak process reduction frequency on the order of about 100 kHz for the Fleng interpreter, and about 400 kHz for the μWAM interpreter. It seems that the μWAM interpreter is much faster but this impression is a bit misleading, since the μWAM interpreter has more different intermediate language instructions to interpret, and the speed is much affected by the order in which these instructions are executed. The optimal order is different for different programs.

This is not faster than current sequential implementations, but the Connection Machine becomes very interesting in the light of its scalability: The speed of the slowest SIMD instruction of our interpreters (reading data from a remote processor allowing collisions), grows as $O(n/log^2 n)$ in the number n of processing elements, so the performance increases faster with n than most other proposed architectures. A doubling of the number of processors of a 65,536-processor Connection Machine would result in a 1.8-fold increase in reduction frequency.

A main limiting factor is read accesses by different processing elements to shared parts of the user's MIMD program, which very roughly represents about half of the total execution time for both interpreters. Although the potential degree of parallelism is very high, the inference frequency of a *single* process is very low. For this reason, it seems important that the Connection Machine's speed, and not only the degree of parallelism, is increased.

1.6 Related work

Papers suggesting the combination of parallel logic programming languages with SIMD architectures include [23, 24, 10, 11, 12, 13, 17, 18, 25, 9].

One of the earliest projects for investigating SIMD execution of logic programming languages is the DADO project [23, 24]. The DADO machine is a special-purpose, tree-structured computer.

Kanada [10, 11, 12] implements OR-parallel Prolog search programs for vector parallel supercomputers by a kind of compilation. He obtains very high performance, but the method can only be applied to a restricted class of programs. We have suggested and implemented committed-choice languages for vector parallel supercomputers, by vectorization of an interpreter [13, 17]. Tatsuguchi has taken up this approach to implement OR- and restricted AND-parallel versions of Prolog [25].

As far as we know, the first paper to suggest implementing committed-choice logic programming languages on the Connection Machine is [17]. Some initial results for a CM-1 simulator were presented in [15].

The translator for translating Flat GHC into the intermediate language Fleng, was inspired by other researchers working on compiling Parlog and Concurrent Prolog into more restricted subsets [4, 22, 2]. The target languages ("Flat" versions) are allowed to have restricted guards. Our approach is different in that we start with a flat form and compile down to a form without any guard goals at all.

Hirata [7] has designed a language called Oc, which is similar to Fleng, but allows translation from Full GHC into Oc. Unfortunately, some features of Oc makes it hard to implement efficiently.

Another paper dealing with logic programming on an SIMD computer is [9]. This approach extends a traditional sequential Prolog with special set and array constructs which can be executed in parallel.

Bawden and Agre [1] has implemented a version of Scheme on the Connection Machine. However, Scheme is not a parallel language, so this implementation is effectively a number of completely separate sequential Scheme interpreters running simultaneously.

Our approach differs from the above approaches in that we assume a general-purpose parallel SIMD architecture rather than a special-purpose architecture with, e.g. support for logic programs, or restricted communication patterns. We do not rely on locality of programs, or on that programs are sufficiently small to fit into local memories, but expect user programs and data to be spread over the global memory space of the parallel computer, in a minimal-grain approach.

1.7 Paper overview

This paper is organized as follows: In section 2, we overview Flat GHC. In section 3, we give some important benefits of SIMD architectures in the context of Flat GHC implementations. Section 4 describes Fleng, the high-level intermediate language and its interpreter. Section 5 describes the WAM-like low-level language and its interpreter. The SIMD instruction set is outlined in section 6. In this section we also describe the Connection Machine and estimate the execution time of the SIMD instructions. Benchmark results are given in section 7, and are discussed in section 8.

2 Flat GHC

The following is a very brief overview of Flat GHC. More details can be found in, for instance, [29, 30].

A Flat GHC program is similar to a Prolog program. One difference is that every clause contains a "commit"-symbol, |, similar to Prolog's "cut", !. The goals to the left of the commit-symbol are called guard goals, and the rest are called body goals. Execution is similar to Prolog, but clause alternatives may be attempted *in parallel*. As soon as the guard of some clause succeeds, this clause

will be selected for further execution of the body, while execution of the other clauses will be stopped.

Head matching and guard unifications are not allowed to bind any variables occurring outside the clause, so execution of guards cannot affect each other. If such an attempt is made, execution of that process must be suspended. This is the main control mechanism of the language.

We illustrate with an example:

```
a(X,Y) :- X = 0  | Y = zero.
a(X,Y) :- X \= 0 | Y = not_zero.
```

The program is called by a query, such as `?- a(17,X)`. This trivial program binds its second argument to the symbol `zero` if its first argument is zero, and to `not_zero` otherwise.

Suppose now that we call the program with a variable as its first argument, `?- a(Z,X)`: The guard goal `X = 0` will then try to *bind* the first argument, `Z`. This cannot be allowed, since it would affect execution of the second candidate clause. Thus, execution of the first clause must suspend. The second clause also suspends, since it cannot know whether the first argument is zero or not, until it becomes instantiated.

Unifications in the body of a clause are allowed to export variable bindings, since there is no conflict between alternatives at that point. Such a binding can cause a suspended process to resume execution. This would happen if for instance the variable `X` in the example above becomes bound by someone else.

Finally, Flat GHC has the additional restriction that guard goals are only allowed to be certain built-in predicates, not user-defined predicates.

3 Benefits of SIMD Execution

Obviously, the synchronous nature of SIMD architectures simplifies global communication. There are other important advantages with SIMD architectures as well, which are not as obvious:

- Mutual Exclusion

  Several processes competing for common resources need to resolve which process should be selected for each resource. A method which works well for an SIMD architecture is if both processes try to write their process id into a memory cell representing the resource. Then they read back the contents of the cell. If the contents are the same as the process' own id, that process is selected.

  In pseudo-formal code this could be expressed as:

```
for all processors i do
  exclude[resource[i]] := i

for all processors i do
  selected[i] := (exclude[resource[i]] = i)
```

- Avoiding Cyclic Variable-variable Bindings

  The problem of avoiding cyclic variable-variable bindings is quite a hard problem for MIMD systems. The problem occurs when there are simultaneous unifications of variables, such as in a query `?- X=Y, Y=Z, Z=X`. If all these unifications are executed simultaneously, the variables could end up being bound in a circle, and an infinite loop will result the next time one of the variables is dereferenced. Locking the variables cannot be done without risking deadlock. A possible solution is to impose an ordering on variables, such as the order of the variables' memory addresses, and only allow bindings to go from "high" to "low" variables. This method still has drawbacks, such as access contention on low variables, and unability to use a relocating garbage collector.

  For an SIMD implementation these drawbacks can be avoided, because it is enough with a *temporary* variable ordering, which is only used during the binding: First, all variables to be unified are dereferenced. Then variables are bound, observing the temporary ordering. It is easy to see that these bindings cannot introduce any cycles, thanks to the previous dereferencing. After the binding, the ordering can safely be forgotten.

- Global Memory Allocation

  SIMD computers in general and the Connection Machine in particular, allow an elegant way of allocating heap storage for processes. The heap is stretched out across processors, so that heap element 0 resides in processing element 0, heap element 1 in processing element 1, etc. In our implementation, the heap consists of two pvars, `CAR!` and `CDR!`. There is another pvar `FREE!`, which indicates whether the corresponding heap cell is occupied or not. Given a pvar `NEED!`, the memory allocation procedure returns the index of some free heap cell in the corresponding position of another pvar `AVAILABLE!`, for all processors in need, and sets the new occupied positions in `FREE!` to `false`.

  This allocation is non-trivial, but can be done very efficiently on the Connection Machine, where it is called `*PROCESSOR-CONS`. This operation is described in an extremely readworthy article to which we refer the reader for more details [21]. Here, it should suffice to say that we can implement this operation as a combination of ten instructions from our SIMD instruction set.

4 The High-level Interpreter

4.1 Fleng

Instead of interpreting Flat GHC directly, we translate it to an intermediate language, Fleng. Although Fleng is not very different from Flat GHC, it is considerably easier to write a compact interpreter for.

A Fleng program is a set of clauses, like Flat GHC clauses, but do not have any guard goals. A difference is that in Fleng, failure is guaranteed not to be propagated until there are no more active processes. This is in fact a possible execution of a Flat GHC program, but Flat GHC does not *guarantee* that failure is delayed until the execution is finished.

Fleng has only three system predicates: `unify`, `compute`, and `call`. An important common feature of all built-in predicates is that it is always possible to detect their termination. This enables Fleng programs to show termination by recursively checking termination of their components.

The unification primitive `unify(R,X,Y)` is special in that that it takes three arguments: `R` will be bound to `true` if `X` unifies with `Y`, but `false` if `X` doesn't unify with `Y`.

`compute(Op,X,Y,R)` computes binary operations indicated by the argument `Op`, which may be bound to one of the following: `+`, `-`, `*`, `/` (arithmetic), `and`, `or`, `xor` (bitwise), `=`, `<`, `sametype` (comparison). `=` and `sametype` allow unbound variables as arguments.

Fleng also has a one-argument meta-call, `call(X)`.

For a more detailed discussion of Fleng and its properties, the reader is referred to [13, 16].

4.2 Translating Flat GHC into Fleng

This section quickly reviews translation of Flat GHC into Fleng. It is a summary of some previous material, but is included here in order to make the paper more self-contained. A detailed description can be found in [14].

The compiler has to convert clauses with guard goals into clauses without. Since most clauses have empty guards, they can be executed unchanged as Fleng clauses.

Most other Flat GHC clauses have only one guard goal, which is a test. A negative result selects one clause, while a positive result selects another clause:

```
p(X,Y) :- X < 0 | q(X,Y).
p(X,Y) :- X >= 0 | r(X,Y).
```

The definition of this predicate can be transformed into Fleng by moving the test out of the clauses. Clauses can then be selected by indexing on the result of the test:

```
p(X,Y) :- less(X,0,R), p1(R,X,Y).
p1(true,X,Y) :- q(X,Y).
p1(false,X,Y) :- r(X,Y).
```

`less` is straightforward to implement in Fleng using `compute`.

Most practical Flat GHC programs can be translated in the mentioned ways. Multiple-goal guards can be translated into Fleng, using the following general idea:

Suppose that we have a Flat GHC predicate p, defined as

```
p(X) :- g1(X) | b1(X).
p(X) :- g2(X) | b2(X).
```

This definition will be converted into the Fleng definition

```
p(X) :- p1(X,N), p2(X,N).
p1(X,N) :- g11(X,N), b11(N,X).
p2(X,N) :- g22(X,N), b22(N,X).
b11(1,X) :- b1(X).
b22(2,X) :- b2(X).
```

The introduced variable `N` is a mutual exclusion variable. It ensures that only the body corresponding to the first succeeding guard is executed.

The compilers task is to convert the guard `g1(X)` into the guard test `g11(X,N)`, where it must be clear that `g11` *cannot export any bindings outside* `p1` except for the mutual exclusion variable `N`. The compiler must similarly convert the guard `g2`. Since guards may only consist of system predicates, it is not very hard for the compiler to make sure that the converted guards will not attempt to export any bindings.

4.3 Mapping on Processors

Fleng programs are stored as Lisp-type S-expressions on the heap, i.e. in two pvars, `CAR!` and `CDR!`. No attempt is made to store multiple copies of programs locally in each processor, since memories are too small, and this would be a great waste of memory. Programs are spread out over the machine.

For each type of process, one process is represented by one processor. About 3-5 data which go along with the process are stored in the same processing element. These data are pointers to the heap, to a trust cell, or an environment record for variable bindings. There is always a mask vector for each type of process saying whether there is a process at that position or not, so that new processors can be allocated by *PROCESSOR-CONS.

4.4 Interpreter structure

This interpreter is an adaptation of the SIMD interpreter for a vector parallel computer described in [18]. Although the interpreter operates on vectors rather than scalars, it is quite similar to interpreters for committed choice languages.

A structure sharing scheme is used, since we want to avoid copying overhead, which is both hard to vectorize efficiently, and which requires much garbage collection overhead.

The interpreter consists of three main phases: An AND-step, an OR-step, and a UNIFY-step. The AND-step forks queries into separate goal literals. With a definition `p(X) :- q(X)` and a query `?- p(Y), r(Y)`, The AND-step would fork this query into separate goal literals, `p(Y)` and `r(Y)`.

The OR-step then takes the goal literals, and forks them into separate candidate processes. For `p(Y)` in this case, there is only one clause, so only one candidate process is created, containing `p(Y)` and the clause `p(X) :- q(X)`.

The UNIFY-step takes a candidate process and matches the caller with the head of the clause. If matching is successful, the body of that clause is generated as a new query, `?- q(Y)`. In unsuccessful, or if another clause has already been committed to, the candidate is dropped.

In order to prevent one very long unification from delaying other processes which have already finished their unifications, unification must be *interruptable*, so that remaining unification steps are postponed until the next interpreter cycle. The same principle applies to these AND-step for long queries, to the OR-step for many candidate clauses, to dereferencing for long variable chains, and to activation of many suspended processes, unless busy waiting is used.

Except for the three main steps, there is also a simple COMPUTE-step which executes the primitive `compute`.

The following is a more detailed, slightly simplified description of the interpreter:

The AND-step:

- The first element of every query is forked as a separate process, and a *trust cell* is created for this process. The rest of the query, if non-empty, is delayed until the next cycle.

The OR-step:

- A candidate clause, if any is left, is combined with a goal literal, and an environment is allocated. Other candidate clauses for this goal literal have to wait until the next cycle.

The UNIFY-step:

- Both arguments are dereferenced one step. If either argument needs further dereferencing, it has to wait until the next cycle.
- If both arguments are list cells, a cell is allocated on a small stack implemented as a linked list. The CDR parts of the lists are put in this cell, the CAR parts become new arguments, and the unification will continue on the next cycle of the interpreter.
- If one argument is a variable, which must not be bound, the process is suspended. For a busy waiting scheme, this just delays the unification until next cycle.
- If one argument is a variable which can be bound, it is bound. If the stack is empty, this process tries to commit. This is done by mutual exclusion using the trust cell allocated during the AND-step. If the stack is not empty, new unification arguments are popped from it, and the unification continues on the next cycle.
- If both arguments are equal constants, the stack is checked in the same way as in the previous step, and execution continues in an analog way.
- Otherwise, the unification failed, and this process is removed.

Body unification, i.e. the `unify` built-in is executed by the same code as that of head matching. The necessary extension is a flag which says whether exporting a binding is allowed or not, and provision for binding the result argument.

Unification is performed sequentially, depth-first, left-to-right. Although we could get some extra parallelism out of unification, the overhead for synchronization of different branches of the same unification is expensive. It also seems that failure or success of unification is usually decided by just a few arguments (the idea of indexing), which speaks in favor of sequential unification. Parallelized unification also has a tendency to create "bursts" of short-lived parallel processes. Serial unification keeps the degree of parallelism more even.

We can see the interpreter as an abstract machine which can execute four different instructions, AND, BUILTIN, OR, and UNIFY. The source operands are the input processes, and the destination operands are the output processes. As we have described it, these steps are executed cyclically. However, nothing prevents us from executing them in a different order. For instance, it is probably useful to execute a few UNIFY-steps in a row.

5 The Low-level Interpreter

5.1 Compilation and the μWAM interpreter

μWAM is similar to WAM, and compilation from Flat GHC into μWAM is quite similar to compilation from Prolog into WAM. A difference is that environments cannot be allocated on a stack but must be records in a heap. The implementation is also much simpler, since there is no backtracking in Flat GHC, so choice points do not exist.

As opposed to the Fleng interpreter, the μWAM interpreter is structure-copy based.

The following is an example of a maximally optimized `concatenate`:

```
concatenate:
        deref(a0)
        switch(a0,sym,num,cons,var)
sym:    eq(a0,nil,empty,fail)
empty:  bind(a2,a1,concatenate)
        jump(succ)
cons:   getlist(a0,t0,a0)
        newvar(t1)
        cons(t0,t1,t2)
```

```
        bind(a2,t2,concatenate)
        mov(t1,a2)
        jump(concatenate)
var:    suspend
num: fail:
        fail
succ:   return
```

We have assumed that either a mode declaration, or program analysis has shown that the third argument will always be unbound when this procedure is called.

The instruction `eq(x,y,label1,label2)` jumps to `label1` if comparison is successful, and to `label2` is comparison is unsuccessful. The instruction `bind(x,y,label)` jumps to `label` if `x` may not be bound, otherwise it binds it.

The argument "registers" `a0-a2` and temporaries `t0-t2` are stored in the process environment. There are a few other instructions for creating and handling such environments, and there are also instructions for arithmetic and for full unification. An important issue is how to avoid having a wide semantic gap between a unification instructions and the much simpler instructions. We have solved this problem by effectively unfolding the first level of unification, and making recursive calls to the unfolded code.

This interpreter has many more different instructions than the Fleng interpreter, but the principle of cycling through the instructions is basically the same, but for this interpreter the order in which instructions are executed is much more important for the total efficiency.

5.2 Mapping on Processors

The heap is represented, and memory allocation is performed in the same way as for the Fleng interpreter. However, programs are not stored in this heap, but in a different pvar. The program counter for each process is an index into this pvar. Programs are *not* stored locally in processors, for the same reason as for the Fleng interpreter.

Process environments are not very large and could be stored in the memory of a processor. This would speed up execution, but at the time of this writing we have not implemented this optimization.

Otherwise, processes map on processors in the same way as for the Fleng interpreter: One process per processor, and a mask pvar which says whether the position is free or not.

6 An SIMD Machine Instruction Set

6.1 Instructions

We do on purpose restrict ourselves to as small an instruction set as possible. Exotic instructions will prevent comparisons with other architectures, and complicate timing estimation. Fortunately, not many instructions are necessary. Each instruction corresponds to one or a few Connection Machine instruction [26]. (All of these instructions do in fact also exist as either a single or a few machine instructions of typical vector parallel supercomputers, e.g. [8].)

Although in this paper we will use timing data for the Connection Machine, results should be qualitatively applicable to any hypercube-based SIMD computer.

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)
add(s1,s2,d,m)
sub(s1,s2,d,m)
mul(s1,s2,d,m)
div(s1,s2,d,m)
and(s1,s2,d,m)
or(s1,s2,d,m)
xor(s1,s2,d,m)
```

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

```
cmpeq(s1,s2,d,m)
cmpne(s1,s2,d,m)
cmplt(s1,s2,d,m)
cmpge(s1,s2,d,m)
```

We assume that `store` described earlier never causes collisions by trying 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.

When we need fully general `load` and `store` operations which can manage collisions, we use the operations `loadcoll(s,x,d,m)` and `storecoll(s,d,x,m)`. The distinction is important, because as we shall see later, the speed of these pairs of instructions differ substantially. `storecoll` introduces non-determinism by storing values in the destination by overwriting, without control over in *which* order the values are written. This operation is useful for arbitration of parallel processes.

We need to be able to count or enumerate all processors with their corresponding mask element true. The `count(d,m)` operation counts the number of masked positions, and fills `d` with this value. `enumerate(d,m)` could be described as

```
tmp := 0
for all processors i do
  if (m[i]) then d[i] := tmp := tmp + 1
```

6.2 Instruction Timing for the Connection Machine

We will now take a look at how our SIMD instruction set matches that of Paris, the Connection Machine parallel instruction set, and the execution time of the instructions. The timings are based on data and formulas obtained from [5, 6, 26, 28, 27].

We assume that all operations are 32-bit operations. This simplifies the estimation, but is overly pessimistic, since 1-bit or 16-bit operations are sufficient in many cases, and the execution time is considerably shorter for bit-narrow instructions. For the time complexity, we assume that the bit-width of operands is of the same order as the number of processors.

- `mov,movs,add,adds,sub,cmp..`: These operations are all of time complexity $O(\log n)$, if we assume that the order of the range of arguments is the same as the number of processors. The execution time for 32-bit arguments is less than 20 μs.
- `mul,div`: It is not quite clear from our sources what the timing of these instructions are, but straight forward coding should produce code which takes approximately 400 ns, considering the time for addition. The speed of these instructions is not very important for our implementation anyway, since they are only rarely executed. The order is $O(\log^2 n)$.
- `enumerate, count`: These instructions are implemented taking advantage of the hypercube structure of the Connection Machine, and execute in just 400 μs. The order is $O(\log^2 n)$.
- `store`: The time for a 32-bit operation is approximately 800 μs. It is a $O(\log^2 n)$ operation. One factor $\log n$ comes from message transmissions being serial. The other factor $\log n$ comes from the maximum distance between processing elements.
- `load`: This is the same as two `store` operations: The first processor sends its own address to the other processor, which returns the required data. The time spent by this $O(\log^2 n)$ operation is approximately 1600 μs.
- `storecoll`: This instruction was slow for the first version of the Connection Machine, CM-1, but performance has been much improved by CM-2. For CM-2, message routers on the way to a destination are able to delete messages bound for the same destination. This makes the time complexity $O(\log^2 n)$, and the execution time is about 1.6 ms.
- `loadcoll`: This operation is more complicated than `storecoll`. How to implement this operation efficiently is a semi-classical problem of parallel computers. CM-1 solved the problem by using sorting as a subroutine, giving the operation a time complexity of $O(\log^3 n)$. CM-2 hardware allows a more elegant scheme called "backward routing," where messages are combined and saved in routers along the way, and the required data are returned by using exactly the same kind of routing backwards. This reduces the time to about 3.2 ms, and the complexity to $O(\log^2 n)$.

7 Results

It is very hard to find suitable benchmarks for parallel implementations. It is important that the benchmarks have predictable behaviour, so that different implementations can be compared. For instance, we would like the degree of parallelism not to fluctuate wildly. For such reasons, we have chosen to use the traditional `concatenate` benchmark. By running several `concatenate` in parallel, and counting the executed instructions, we have found approximate values for the different classes of instructions, as shown in the table below. The values given are per processing element and process reduction for 65,536 parallel processes on a 65,536-processor Connection Machine.

The peak process reduction speed for the Fleng interpreter would be approximately be $65,536/0.606$ Hz $\approx$ 108 kHz.

Instruction group	Frequency	Total time per reduction [ms]
`mov, add, etc`	1911	38
`mul, div`	0	0
`enumerate, count`	87	35
`store`	154	123
`load`	59	94
`storecoll`	11	18
`loadcoll`	93	298
Total	2,315	606

Benchmark results for the Fleng interpreter.

There are no calls to `compute` in `concatenate`, so `mul` and `div` are not called at all.

Instruction group	Frequency	Total time per reduction [ms]
`mov, add, etc`	776	16
`mul, div`	0	0
`enumerate, count`	51	20
`store`	14	11
`load`	11	18
`storecoll`	0	0
`loadcoll`	32	102
Total	884	167

Benchmark results for the μWAM.

For this benchmark and the μWAM interpreter, no mutual exclusion is necessary, so the count for `storecoll` is zero.

The peak process reduction speed for the μWAM interpreter would be approximately be $65,536/0.167\,\text{Hz} \approx 392\,\text{kHz}$.

7.1 Analysis and Discussion

The simulation data shows clearly that execution time is dominated by `loadcoll`. Most of the calls to `loadcoll` turns out to be the interpreters reading user programs. It is tempting to try to store programs locally in processors to speed up such operations. However, even if the time for `loadcoll` could be reduced to zero, the speed would only increase by a factor of less than two, which hardly makes such a solution worthwhile.

The speed figures we have obtained are for ideal circumstances. It is clear that for a less homogeneous mix of predicates, the performance would be less, may be not so much for the Fleng interpreter as for the μWAM interpreter: Suppose that some process would like to execute the instructions a, b, c, d, e, f, in this order, and that some other process would like to execute them in the opposite order, f, e, d, c, b, a. This will reduce the total speed to most half, in whichever order we execute instructions. For the Fleng interpreter this effect is not so pronounced, since there are only four "instructions'."

Even if the performance of the μWAM interpreter for this reason sometimes may fall to the level of the Fleng interpreter, it may have an advantage in simplifying garbage collection. In this paper, we have not discussed garbage collection, which is obviously an important topic. It does seem that garbage collection is well suited to SIMD execution, since it garbage collection is of parallel nature, and consists of a large number of relatively simple operations. Garbage collection for SIMD architectures is discussed in [19].

At a peak reduction frequency of a few 100 kHz, the Connection Machine is not faster than current sequential implementations. However, the Connection Machine becomes much more interesting considering its scalability: Since the speed of the slowest SIMD instruction of our interpreters (reading data from a remote processor allowing collisions), grows as $O(n/log^2\,n)$ in the number n of processing elements, the performance also increases at this rate. A doubling of the number of processors of a 65,536-processor Connection Machine would result in a 1.8-fold increase in reduction frequency.

This discussion has all the time assumed that programs possess very much parallelism. For strictly sequential programs, performance is abysmal. Thus, a suitable future direction may be to combine a fast sequential, or low-degree parallel front-end computer, with a massively parallel SIMD computer as a back-end.

The approach we have described should be applicable to other committed-choice languages, such as Parlog, as well. However, languages which rely on very complex non-interruptable primitive operations, like atomic unification, do at least for now seem less suitable for SIMD implementation.

8 Conclusions

The results of the simulation show that implementing Flat GHC on the Connection Machine is not a free lunch. The speed is comparable to that of mainframes. However, the advantage with the hypercube SIMD approach is its unique scalability. This assumes that programs executed contain massive parallelism - if not, performance will be very low. The best approach for the future may be to combine a sequential processor with a massively parallel processor as a back-end.

9 Acknowledgments

We are very much indebted to Ken Kahn, who really went out of his way to arrange a visit for us at Xerox Parc, introducing us to Connection Machine wizards and to the real machine, and help us around in all kinds of ways. Without this visit, which became possible entirely thanks to Ken, our paper would have been much less interesting (or at least, much more boring). We are grateful to Xerox Parc for generously sponsoring our visit there.

We are also grateful to Charles Elkan from Cornell for very valuable discussions of various CM topics, which especially inspired work on the μWAM implementation. We learned much about the Connection Machine from John Lamping at Xerox Parc, and Donna Fritzsche from Thinking Machines, who spent a long time patiently explaining the machine and *Lisp to us.

We have benefitted 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, especially Kazunori Ueda.

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

References

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

[2] Codish,M. and Shapiro,E.: *Compiling OR-parallelism into AND-parallelism.* In Shapiro, E. (ed): Proc. 3rd Int. Conf. on Logic Programming, London. July 1986. p. 593-599.

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

[4] Gregory, S.: *Parallel Logic Programming in Parlog* Addison-Wesley, 1987.

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

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

[7] Hirata, M.: *Self-description of the Parallel Programming Language Oc.* Computer Software, No. 3, Vol. 4. September 1987. (In Japanese)

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

[9] Kacsuk, P. and Bale, A.: *DAP Prolog: A Set-oriented Approach to Prolog.* Computer Journal, Vol. 30, No. 5. 1987. p 393-403.

[10] Kanada, Y.: *High-speed Execution of Prolog on Supercomputers.* In Proc. 26th Programming Symp., Information Processing Society of Japan. 1985. p 47-56. (In Japanese)

[11] 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, 87-PL-12. July 1987. p 1-10. (In Japanese).

[12] Kanada, Y., Kojima, K., and Sugaya. M.: *Vectorization Techniques for Prolog.* In Proc. Int. Conference on Supercomputing. St Malo, France. August 1988.

[13] 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. June 1986. p 209-216. Also in Wada, E.(Ed.): Logic Programming '86, Springer LNCS 264. p 170-179.

[14] Nilsson, M. and Tanaka, H.: *Converting FGHC Clauses with Guards into Clauses without Guards.* In Information Processing Soc. of Japan Workshop on Programming Languages, 88-PL-17. July 1988. (In Japanese). Section 17-3.

[15] Nilsson, M. and Tanaka, H.: *SIMD Architecture and Superparallel Logic programming.* In Information Processing Soc. of Japan Workshop on Computer Systems, 88-ARC-71. July 1988. Section 71-16.

[16] Nilsson, M. and Tanaka, H.: *The Art of Building a Parallel Logic Programming System.* In Tanaka,H. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, June, 1987. p 155-163. Also in Furukawa, H., Tanaka, H., Fujisaki, T. (Eds.): Logic Programming '87. Springer LNCS 315, 1988. p 95-104.

[17] 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.

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

[19] Nilsson, M. and Tanaka, H.: *Graph Algorithms for Supercomputers.* To appear in Proc. Int. Computer Symposium, Taipei, Taiwan. December 1988.

[20] Shapiro, E.: *A Subset of Concurrent Prolog and its Interpreter.* ICOT Technical Report TR-003, February 1983.

[21] Steele, G.L., Jr. and Hillis, W.D.: *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing* In Proc. 1986 ACM Conf. Lisp and Functional Programming, Cambridge, Massachusetts. August 1986. p 279-297.

[22] Sterling, L. and Codish, M.: *Pressing for Parallelism: A Prolog Program Made Concurrent.* J. Logic Programming, No. 1, 1986. p. 75-92.

[23] Stolfo, S.J.: *On the Limitations of Massively Parallel (SIMD) Architectures for Logic Programming.* In Proc. US-Japan AI Symp. 1987. ICOT, Tokyo, Japan. December 1987. J. Logic Programming, No. 1, 1986. p. 75-92.

[24] Stolfo, S.J. and Shaw, D.E.: *DADO: a tree-structured machine architecture for production systems.* Proc. National Conf. Artificial Intelligence, Carnegie-Mellon University. August 1982.

[25] 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).

[26] *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Corporation, Technical Report 87-7. April 1987.

[27] *Personal communication.* Thinking Machines Corporation, August 1988.

[28] Tucker, L.W., and Robertson, G.G.: *Architecture and Applications of the Connection Machine.* In IEEE Computer. August 1988. p 26-38.

[29] Ueda, K.: *Guarded Horn Clauses.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo. July 1985. p 225-236. Also in Wada, E.(Ed.): Logic Programming '85, Springer LNCS 221. p 148-167.

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

[31] Warren, D.H.D: *An Abstract Prolog Instruction Set.* Technical Note 309. Artificial Intelligence Center, SRI International, 1983.