A FLAT GHC IMPLEMENTATION FOR SUPERCOMPUTERS

MARTIN NILSSON and HIDEHIKO TANAKA
The Hidehiko Tanaka Lab., Dept. of Electr. Engineering,
The University of Tokyo, Bunkyo-ku, Hongo 7-3-1, Tokyo 113

ABSTRACT
We describe an implementation of Flat GHC for vector parallel supercomputers. We first convert GHC programs to a simpler form, which does not contain any guard goals, and where failure of a goal does not implicitly affect other goals. We then execute the programs by a vectorized interpreter to attain high degrees of parallelism. For the Hitachi S-820/80 supercomputer, the implementation performs about 1.1 million process reductions per second. The vectorization ratio of the interpreter is close to 100%. The speed-up by vectorization is comparable to the speed-up of numerical programs.

1 Introduction

Although much research is being done on computers for parallel execution of logic programming languages, very little work has been done so far on logic programming for supercomputers. Supercomputers have traditionally been considered useful only for high-speed numerical computations. The reason for this is that they, almost exclusively with SIMD architecture, have severe programming constraints: Parallel programs must be expressed as vector operations, or as vectorizable loops. Earlier generations of supercomputers, such as Star-100 and Cray-1 have restricted sets of vector instructions, which make them unsuitable for non-numeric applications [1]. However, the development of supercomputers has been very fast, and more recent supercomputers, such as Cray-XMP and Hitachi S-820 [13],[8], have so much improved power, that they are useful also for non-numerical applications.

One of the interesting features of logic programming languages is that interpreters for these languages often can be expressed as very compact loops. Our research is based on the idea that *if we can express the language interpreter as one or a few vectorizable loops*, we might get a very efficient supercomputer implementation.

We initially chose the language GHC [22], which is a parallel logic programming language of committed-choice type. 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, eg. [3]. This language turned out to be a bit too complex for direct implementation, so we convert it into a simpler language, Fleng [15].

There is an empirical guideline for Prolog implementations which says that the inference frequency (LIPS number) is very approximately $f/1000$ for an interpreter,

where f is the number if machine instructions per second (MIPS number). If we take the peak performance as the MIPS number for the supercomputer, we obtain a "dream" value of about 3 MHz inference frequency for the S-820 supercomputer. Since this computer has 12 pipelines, several of which are dedicated to multiplication, etc., which is not useful for our implementation, the actual number of useful pipelines is something like four to eight. This reduces the dream value to 1-2 MIlz.

Our Fleng interpreter comes quite close to this value, by performing about 1.1 million process reductions per second on the S-820. The cost of parallelism in this implementation is very cheap: One extra degree of parallelism costs about 80 memory bytes. Almost 100% of the inference loop are vectorizable instructions. The speed-up by vectorization is comparable to that of numerical programs.

The main purpose of the work reported is to show that vector parallel execution is a viable candidate for high-speed execution of logic programming languages.

After mentioning related work in the next section, we will briefly describe GHC and Fleng in section 3, outline vector processor execution in section 4, explain the structure of the vectorizable interpreter in section 5, and discuss the results of an implementation for the S-820 supercomputer in section 6.

2 Related work

The first person we know of who has suggested implementing a logic programming language on supercomputers is Kanada [11],[12]. He has compiled several OR-parallel Prolog search programs into Fortran, and reached large degrees of parallelism this way. However, this compiled approach works only for OR-parallel search programs. The compilation is hard to do automatically. We suggested instead implementing an AND-parallel committed-choice language as a tight loop vectorizable interpreter [14]. This would allow high degrees of parallelism for programs in general.

Tatsuguchi [21] has used this approach to implement OR-parallel and restricted AND-parallel interpreters for a vector parallel supercomputer. These implementations are based on a structure copying scheme, and a large part of the code is scalar, so the vectorization speed-up has been slight. Our current implementation is based on structure sharing, to avoid overhead for copying and minimize overhead for garbage collection. There is no restriction on what kind of programs can be executed.

Our intermediate language Fleng is derived from Kernel Parlog [4], and GIIC. It is similar to Oc [6],[7], in that Oc does not have any guard goals either. However, Oc was designed with logical purity in mind, and is very hard to implement efficiently. Fleng was, on the contrary, designed to be efficiently implementable.

Other papers dealing with the compilation of committed-choice languages into lower-level languages are [2], [18], [4]. We go a bit further, by not allowing *any* guard goals in the target language.

Kacsuk and Bale [10] proposes an implementation of Prolog extended with built-in predicates for manipulating data residing in an SIMD computer. However, in this implementation, the program itself is not executed by the vector processor.

Stolfo has also described implementation of Prolog on a special purpose SIMD com-

puter [19], [20]. This implementation uses larger grain size than we do, about the size of one inference, while we use a grain size of a single (vector) machine instruction.

3 Flat GHC and its compilation into Fleng

GHC compared to Prolog

Full GHC is implementable, but the generality is costly. Flat GHC, which is a subset of Full GHC, is generally considered to be sufficiently powerful. Currently, the name "GHC" has come to normally refer to Flat GHC. In this paper, we will use the name GHC in this sense.

A Full GHC program is quite 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 also similar to that of Prolog, but the most important difference is that all clause alternatives may be attempted *in parallel*, while in Prolog, they must be tried sequentially. However, as soon as all the guard goals of some clause have succeeded, this clause will be selected for further execution of the body, while execution of the other clauses will be stopped. In this way, commitment works as a mutual exclusion mechanism.

In order to allow parallel execution of clause candidates, we must ensure that they do not affect each other. In other words, head matchings and guard goals may not bind variables which are accessible from the outside. If such an attempt is made, execution of that process must be suspended.

We illustrate this with a simple 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 it 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 this 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.

Note that although guards cannot export variable bindings, body goals can, since there is no conflict between alternatives at that point. If the variable `X` in the example above subsequently becomes bound from somewhere else, the suspended processes wake up and resume execution.

Flat GHC is the restriction of GHC, where all guard goals belong to a predefined set of built-in primitives. The reason for this restriction is that if user-defined goals are allowed in guards, the overhead for making sure that no illegal binding exports are made becomes very expensive.

Fleng

A Fleng program is a set of clauses, looking like Prolog clauses, with a head part and a body of goals.

```
H :- G1, G2, ...,Gn.
```

A program is executed by giving it a set Q of goals. All goals are executed independently, in any order. A goal G is executed by removing it from Q, and matching it with the head of all clauses in the program. For the first clause with matching head, that clause is committed to, and the body goals of that clause are added to Q. Matching is like unification, but variable bindings may not be made in G (exported) during the matching. If a match of a variable in G with a non-variable, or an unbound variable from G is attempted, this matching is suspended, and resumes if the variable becomes bound.

As opposed to GHC, the execution of a goal is not associated with success, failure, or any logical truth values. *New goals are essentially "spawned" out and executed. Unmatched goals do not produce any failure, but disappear without any side-effects.* This is one property which makes Fleng different from all other logic programming languages we know of. The reason behind this is that in flat committed choice languages, the failure propagating mechanism is more of a nuisance than an asset. We decided to skip it, because if we *do* want logical truth values, they can be passed by parameters of the goals. For instance, if we have two goals `g(X,R1)` and `h(X,R2)`, where `R1` and `R2` are bound by predicates `g` and `h` to the appropriate truth values, we can define a new predicate `p(X,R3)` to be the logical conjunction of `g` and `h` by:

```
p(X,R3) :- g(X,R1), h(X,R2), and(R1,R2,R3).
```

where

```
and(true,true,R) :- R=true.
and(false,_,R)   :- R=false.
and(_,false,R)   :- R=false.
```

Fleng has three built-in 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.

- `unify(R,X,Y)`
  `R` will be bound to `true` if `X` unifies with `Y`. `R` will be bound to `false` if `X` doesn't unify with `Y`. Bindings which are done during this unification are not undone, even if the arguments are found be ununifiable. The definition `X = Y :- unify(R,X,Y)` is often useful.

- `compute(Op,X,Y,R)`
  The argument `Op` represents a binary operation and may be bound to one of the following: `+`, `-`, `*`, `/` (arithmetic), `and`, `or`, `xor` (bitwise), `=`, `<`, `sametype` (comparison). `=` and `sametype` allow unbound variables as arguments. We can define `var(X,R) :- compute(sametype,X,Y,R)`.

- `call(X)`
  This is the metacall primitive. It does not need a result parameter, since the result is supposed to be reported through a subterm of `X`.

For a more detailed discussion of Fleng and its properties, the reader is referred to [14] and [15].

Compiling Flat GHC into Fleng

A closer look at practical GHC programs [3], reveal that as much as 80% of typical GHC clauses have only the guard goal `true`, and most of the rest (about 15%) have only a single guard goal. Furthermore, guards are very often mutually exclusive, as in our example in the previous subsection. These facts suggest that a further simplification of guards is possible. This would certainly be welcome, since guard goals is one of our biggest reasons for implementation headache. In particular, guard goals make it hard to make the interpreter cycle compact enough for efficient SIMD execution.

We will start by showing some simple examples of how some typical GHC clauses are translated into Fleng.

The point of GHC to Fleng compilation is translation of a predicate definition including guard goals into a definition without.

As we have mentioned, most clauses have only the trivial guard goal `true`. In this case, they can be executed unchanged as Fleng clauses.

Most other 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).
```

If we want to avoid the intermediate step of `p` calling `p1`, we can expand a call `...p(X,Y),...` directly into `...less(X,Y,R), p1(R,X,Y),...`. The Fleng predicate `less(X,Y,R)` could be defined as

```
less(X,Y,R) :- evaluate(X,A), evaluate(Y,B), compute('<',A,B,R).
```

where `evaluate(X,R)` evaluates the expression `X` and binds the result to `R`. `evaluate` is straightforward to implement in Fleng using `compute`.

Most practical 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 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 `g11(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.

More details on this translation can be found in [17].

4 Supercomputers and Vector Parallel Execution

Supercomputers require (Fortran) programs that are expressed as vectorizable operations for efficient execution, i.e. DO-loops. A typical DO-loop

```
      DO 10 I = 1,512
        operation A
        operation B
        operation C
10    CONTINUE
```

is then executed by first executing operation `A` for all values of `I`. Then, operation `B` is executed for all values of `I`, and finally `C`. Temporary results are kept in *vector registers*, which have a function analogous to that of registers of a conventional computer. Since vector registers have a limited length, usually 64 to 256 for current supercomputers (256 for S-820), operation A cannot be performed for all values of `I` at a time. For this reason, operation `A` is performed for `I` from 1 to 256, then `B` and `C`. After that, `A`, `B`, and `C` are performed for `I` from 256 to 512.

Although vector registers are 256 elements long, this does not mean that this many operations are executed simultaneously: The individual operations are *pipelined*, although the pipeline *pitch*, i.e. the delay between entering two elements in the pipeline, is very short (4 ns for S-820). The main factors deciding the speed of the computer is this pipeline pitch, and the number of pipelines. Several pipelines allow either a vector operation to finish faster, or several operations to be run in parallel pipelines (such as `A`, `B`, and `C` in the example). Current supercomputers have four to sixteen pipelines.

The are three kinds of vector operations which are crucial for implementations of symbolic processing languages on supercomputers: *List vector instructions*, *masked instructions*, and *vector compression*.

List vector instructions are sometimes called *index vector*, *vector indirect*, *scatter*, or *gather* instructions. We shall avoid the word "gather," since this is sometimes instead used for compression operations.

List vector instructions correspond to indirect instructions of conventional computers. They are necessary for pointer manipulation. When vectorizing a Fortran DO-loop, list vector instructions are used where indices of arrays are themselves array references:

```
      DO 10 I = 1,4711
        A(B(I)) = C(I)
 10   CONTINUE
```

The ability to mask out operations is necessary for vectorizing loops which contain conditional statements:

```
      DO 10 I = 1,4711
        IF (A(I) .GT. 17) B(I) = C(I)
 10   CONTINUE
```

For this purpose, supercomputers have special *mask vectors*, which can be set or reset depending on the result of vector compare instructions. Compress instructions take all elements of one vector which have a mask bit set, and compresses them into contiguous locations in another vector. This operation corresponds to a DO-loop

```
      DO 10 I = 1,4711
        IF (A(I) .GT. 17) THEN
          N = N + 1
          B(N) = C(I)
        ENDIF
 10   CONTINUE
```

This kind of operations is important for e.g. memory compaction after garbage collection, and efficiently extracting elements satisfying certain conditions from a queue.

To be vectorizable, loops should not contain any subroutine calls. Jump instructions cannot generally be used, but are allowable, if they can be converted into IF-THEN-ELSE constructions by the compiler's data flow analysis.

When we want to do different operations a vector depending on the result of a test, we have two options: One is the straight forward approach using mask bits:

```
      DO 10 I = ...
        IF condition THEN
          A(I) = ...
        ENDIF
 10   CONTINUE
```

The other way is to first collect indices in separate, compressed vectors, and then perform the operations in a separate loop:

```
      DO 10 I = ...
        IF condition THEN
          N = N + 1
          COLLECT(N) = I
        ENDIF
10    CONTINUE

      DO 20 I = ...
        A(COLLECT(I)) = ...
20    CONTINUE
```

For our implementation, we found the latter approach to usually be the fastest. The tendency towards building supercomputers with multiple pipelines also favors this approach, as it becomes much more complicated to do pipeline "skips."

More details on supercomputers can be found in, e.g. [9].

5 Interpreter Structure

For vectorization, as much code as possible 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 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.

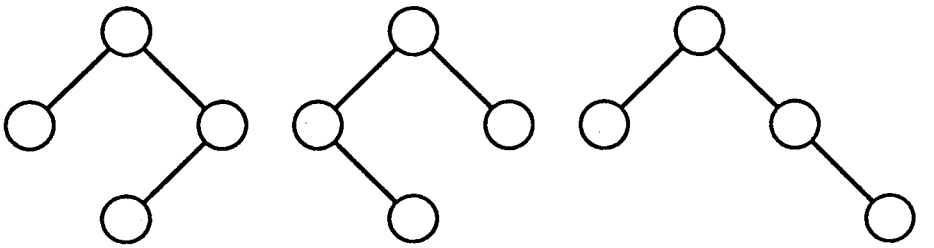
The general structure of the interpreter is not unlike that of a traditional Prolog interpreter, and even less different from that of a Flat GHC or Fleng interpreter [15]. The main difference is that all processes who want to do the same operation are collected, and their data are operated on as vectors, where the N-th elements are data belonging to process N.

Data Structures

To each process belongs a record of data, consisting of 3-5 pointers, such as pointers to clauses, literals, environments, symbols, variables, etc. Almost all data are

accessed through pointers, which makes it important that list vector operations can be performed efficiently.

Since our implementation is structure-sharing, data need not be copied. and programs (and skeletons) reside in Lisp-style traditional CAR and CDR vectors. These vectors are fixed during execution.



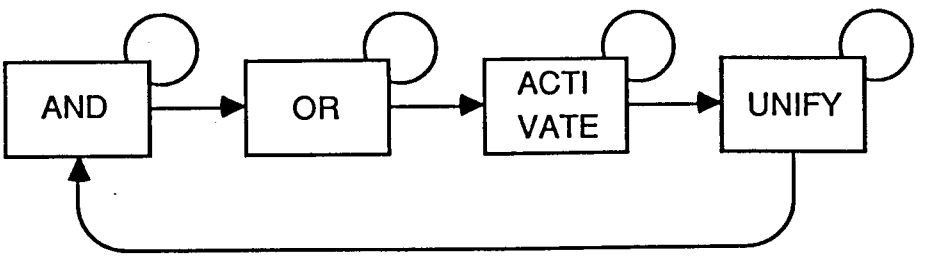
One reason why it is very practical to have binary trees as basic data structures is that although trees may have different shapes, usually the operations at the nodes are the same, so we can efficiently vectorize operations on trees, if the grain size is a node.

Program Structure

All operations should be made *real-time*, i.e. finishable in bounded time: It is unsuitable if 9999 processes wait for one very long unification. Such non-real-time operations as unification should be split into smaller real-time steps.

The interpreter has four different operations which are non-real-time: Unification (including dereferencing); process activation, since several processes might be restarted by one activation; AND-reduction, since there may be several goals in a body; and OR-reduction, since there may be several clauses defining a predicate.

Our interpreter consists of four blocks, each a loop on its own. Each block corresponds to one non-real-time operation, so there is an AND-, OR-, ACTIVATE- and a UNIFY-block. A block takes a queue of processes as its input and produces two



new queues of processes: Processes which are ready for the next block, and processes which should be fed back into the current block. UNIFY also adds processes to the special queue for activation.

- AND
  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 meta-call and arithmetic built-ins.
- OR
  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 (body) unification.
- ACTIVATE
  The ACTIVATE block prepares activated goals for re-unification.
- UNIFY
  The UNIFY block handles both passive (head) and active unification. It handles dereferencing, variable binding, and commitment. Successful passive unifications generate new goals for the AND block.

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.

In an optimized version of the interpreter, we do actually modify this scheme slightly to accommodate indexing: We start by doing some serial unifications, and if this works, we continue doing parallel unifications.

Mutual exclusion of processes are necessary at exactly two points in the code: One is for binding variables, and another is for committing clause candidates. These operations are 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.

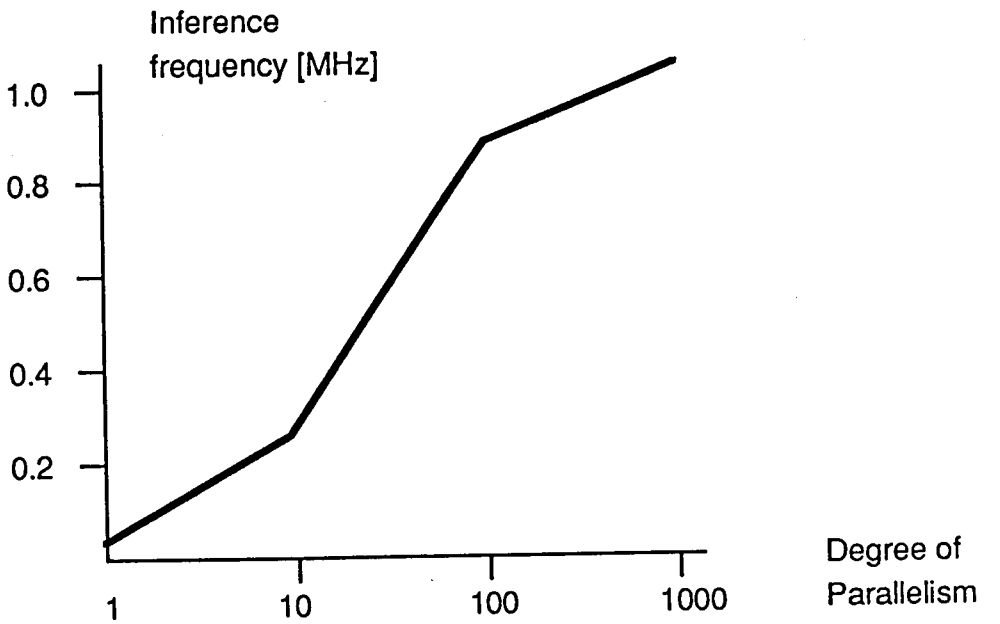
The code for mutual exclusion looks in principle like this (PID stands for "Process ID"):

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

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

6 Results and Discussion

We have implemented two versions of the interpreter. The first interpreter tries very hard to share code, for instance, for head and body unification. The other interpreter has separate code for head and body unification. It also uses a kind of indexing, and some other minor optimizations. The latter version is thus almost twice as large, but runs at about twice the speed of the first version. For a simple benchmark,



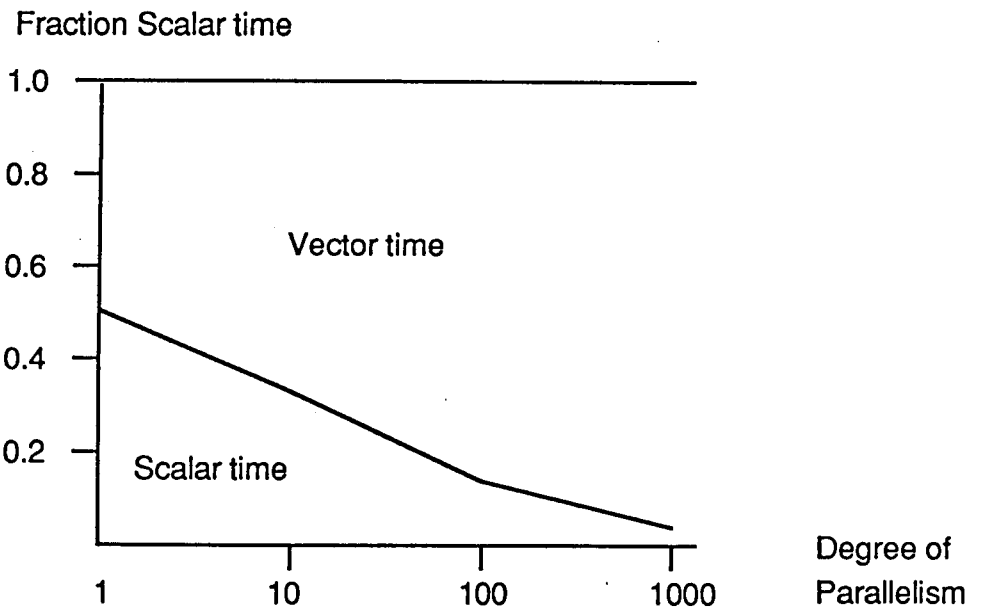
consisting of a number of parallel **concatenate** calls, run on the Hitachi S-820/80, the optimized interpreter achieved approximately 1.1 million process reductions per second, which makes our system the perhaps fastest parallel logic programming interpreter today, and comparable to the fastest compiled systems. The degree of parallelism for this benchmark was 1000. The reduction frequency still increases for higher degrees of parallelism. The cost per degree of parallelism is 80 bytes, i.e. 20 positions of queue storage.

The source code uses conditional macro definitions to be compilable both into C for development on a Sun-3, or, using the Ratfor preprocessor, into Fortran 77, for execution on the supercomputer. This Fortran program is completely portable, except for vectorization pragmas, which are used for guiding the vectorizing compiler. The interpreter cycle contains 20 loops for the non-optimized version. Four are for AND-processing, four are for OR-processing, one is for ACTIVATE-processing, and the remaining eleven are for UNIFY-processing. The optimized version approximately doubles these numbers.

The ratio of vector processor time to scalar processor time goes up to about 20:1 for 1000 parallel processes. This shows that the processing time is clearly dominated

by vector processing.

Other SIMD architectures, such as the Connection Machine [5], seem to be suitable for similar kinds of implementation [16]. We are currently working along these lines, as well as compiling Fleng further down into bytecode, which can be interpreted by an even simpler interpreter.



7 Conclusions

We have demonstrated that a vector parallel architecture can be efficiently used for execution of general-purpose parallel logic programming languages such as GHC, with speeds matching that of the fastest compiled implementations. Although another way (currently much cheaper than buying a supercomputer) to obtain a large speed-up is by compilation, it should be noted that this speed-up is *fixed*, while the speed-up with our approach is *scalable* with the number of pipelines and the pipeline pitch of the vector parallel architecture.

For this reason, we believe that vector parallel architectures should be considered a serious alternative for execution of logic programming languages with very high degrees of parallelism.

8 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 Hirata-san, Kanada-san, Tatsuguchi-san, and Ueda-san. Thanks to Hamanaka-san, who taught us lot about supercomputer hardware. We also owe much to Maruyama-san, Ogino-san, and all the patient people of the Computer Centre, without whose heroic efforts we would never have made it alive, on the traumatic odyssé through occasionally working networks, batch control cards, VOS3 (read JCL), and the obscure compile-parameter-parameters of the local Fortran.

References

[1] Bucher, I.Y.: *The Computational Speed of Supercomputers.* In Proc. ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems. August 1983, p 151-165.

[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] Hirata, M.: *Self-description of Oc and its Applications.* In Proc. Second National Conf. of Japan Society of Software Science and Technology, 1985. p 153-156. (In Japanese)

[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] Hwang, K. and Briggs, F.A.: *Computer Architecture and Parallel Processing.* McGraw-Hill Computer Science Series. 1986.

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

[11] 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)

[12] Kanada, Y.: *The Challenge of 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).

[13] 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)

[14] 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.

[15] 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.

[16] 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.

[17] 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).

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

[19] 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.

[20] 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.

[21] 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).

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