5H-4 A GHC Implementation for Supercomputers

Martin Nilsson and Hidehiko Tanaka *
The University of Tokyo

Abstract

We will describe an implementation of Flat GHC for vector parallel supercomputers. We first compile GHC programs to a simpler form, which does not contain any guard goals. We then execute the programs by a vectorized interpreter to attain high degrees of parallelism.

1 Introduction

We are interested in parallel languages for symbolic data processing on supercomputers. We have been attracted to logic programming languages, because of their relative simplicity and expressive power. We are interested in supercomputers, because of the potentially high degrees of parallelism, and the scalability of the architecture.
We have initially chosen the language GHC [4], 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, e.g. [1].
In order to compactify the central interpreter loop as much as possible, we first compile GHC down into a low-level logic programming language, called Fleng [2],[3]. We then execute Fleng code on the supercomputer by a tightly coded interpreter.
In this paper we briefly describe the compilation of GHC into Fleng, design details of the supercomputer Fleng interpreter.

2 Flat GHC and Fleng

A Fleng clause has a form similar to a GHC clause, but there are no guard goals. Only the head is used for testing.
The goals in the body of a Fleng clause, as opposed to GHC goals, may only return information about their state by binding shared variables. This means that a Fleng body is executed by just "forking" the body goals and execute them independently.
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 succeding 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`.

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.

3 Vectorization

The are two radical ways of vectorizing the interpreter loop: One is to turn every scalar in the loop into a vector, and to make one, big loop of the whole interpreter. We could call this *vertical* vectorization. Another way is to turn every single instruction into a loop. This could be called *horizontal* vectorization.

Completely vertical vectorization is not practically possible unless the loop is short. It also tends to waste execution time. On the other hand, complete horizontal vectorization requires much overhead for preparing vector execution. The ideal vectorization model is somewhere in between these two methods, where the loop size is optimized to keep the supercomputers pipelines/interconnection network at maximum throughput. With current supercomputers, this means bias towards horizontal vectorization.

*The Tanaka Lab., Dept. of Electr. Engineering, Univ. of Tokyo, Bunkyo-ku, Hongo 7-3-1, TOKYO 113

4 SIMD Programming Restrictions

In order to obtain any vectorization speed up, we must make sure that practically *all* code in the interpreter loop is vectorizable. In order to be vectorizable, it obviously may not contain any jump instructions or subroutine calls. Another restriction is that we cannot use linear stacks, since at least one stack would be necessary for every vector element, which would be quite impossible for memory reasons.
The implementation has to be inherently heap-based, allowing processes to allocate memory from a common pool.
All operations has to be made *real-time*, i.e. must be finished in bounded time. Thus, for instance, dereferencing, which is non-real-time, must be split into real-time dereferencing steps. The interpreter has exactly 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; OR-reduction, since there may be several clauses defining a predicate.

5 Fleng Interpreter Structure

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 UNIFY-block.
Each 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 built-ins for arithmetic.

- 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 unification.

- ACTIVATE
  The ACTIVATE block prepares activated goals for re-unification.

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

6 Implementation Details and Discussion

The target language of the implementation is Fortran. In order to make development and debugging reasonable, the source code uses macro definitions extensively, to be compilable both into C, or, using the Ratfor preprocessor, into Fortran.
Unification dominates the code. Initially, we thought that code for mutual exclusion (variable binding and commitment) would be critical for performance, but it now seems different: Variables are bound at exactly one point in the code, and committing is also done at only one point, representing rather little overhead.
Not to get caught in too much idiosynchratic detail of a particular computer's architecture, we cannot go to a much lower implementation language than Fortran. The absolute efficiency of the implementation will thus necessarily be severly limited by the vectorizing Fortran compiler. Despite this, we can still compare relative speed-up and degree of parallelism for vectorized vz. sequential execution on the same computer, in the same language.

References

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

[2] 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 printed as Springer LNCS 264.

[3] 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.

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