A Parallel Graph Reduction Machine

Mike Joy

Department of Computer Science,
University of Warwick,
COVENTRY,
CV4 7AL,
UK.

Tom Axford

Department of Computer Science,
University of Birmingham,
BIRMINGHAM,
B15 2TT,
UK.

10 September 1992

ABSTRACT

In order to facilitate research and teaching into parallelism in functional languages, the authors have implemented a simulator for parallel graph reduction machines. The simulator is capable of modelling both shared and distributed memory machines, and both strict and lazy evaluation strategies. This paper describes the simulator, together with some results achieved with it.

1. Introduction

Implementations of functional languages have often been criticised on account of their tendency to be slow compared with implementations of imperative languages. Parallel machines are one method by which the speed of functional programs can be increased, and is an area of vigorous investigation. The authors are algorithms for distributing sections of functional programs about processors, together with the associated high-level constructs and data-structures. We required a mechanism to test such ideas, and verify their performance on "real" machines, prior to implementing them on a hardware platform.

We also required a tool for use in a course in parallel functional language implementations. It was therefore decided to write a simulator for such a machine.

There were a number of criteria which had to be satisfied:

1) The user interface must be simple and straightforward to use.
2) The simulator must be able to mimic various types of machine, in particular both shared and distributed memory architectures.
3) Varying network topologies must be catered for.
4) The simulator must run fast, so that "real" programs can be run on it (within the limitations which must apply to any simulator).
5) Both strict and lazy evaluation strategies must be programmable.
6) Finally, the simulator must be coded in such a way that it can with reasonable ease be ported to a parallel architecture.

In this paper we describe the machine and indicate some of the preliminary results achieved with it.

2. Design Decisions

The computer resources available to us were a network of Suns running the UNIX® operating system. Coding has therefore been performed in C, and we have stuck rigidly to the ANSI standard to ensure that it is in fact portable.

More problematic was the choice of functional language that the machine would accept. Initially we implemented FLIC [4] together with translators from high-level languages to FLIC, so that we would have the benefits of a variety of sophisticated programming tools at our disposal. This proved to be infeasible, partly due to the extra programming necessary to implement the translators (most high-level functional languages such as Miranda ™ [6] and Haskell [3] are big languages, and to implement well a *full* translator would have taken too much time). It also turned out that the output from such a translator initially produced very inefficient FLIC code, and substantial work would need to be done to yield reasonably optimised FLIC.

We therefore designed a language (called *Ginger*) which sits somewhere between the low-level FLIC and high-level Miranda or Haskell. Ginger is a heavily sugared lambda-calculus, with numbers, characters, lists and booleans as data types; standard number and list operators are included, where appropriate as infix operators. In common with various other functional languages, each infix operator has a prefix form, denoted by the infix operator enclosed in parentheses. Ginger does not support user-defined data types; these were felt unnecessary for investigating parallelism, but will be implemented later if time allows. Pattern-matching is not allowed, all user-defined functions are prefix (curried), and to define a function one can use either the construct

```
f x y = ...
```

or

```
f = \x \y ...
```

Local definitions using either

`let` *definitions* `in` *expression*

or

expression `where` *definitions*

are allowed. A Ginger program will look very similar to an equivalent one in Miranda or Haskell.

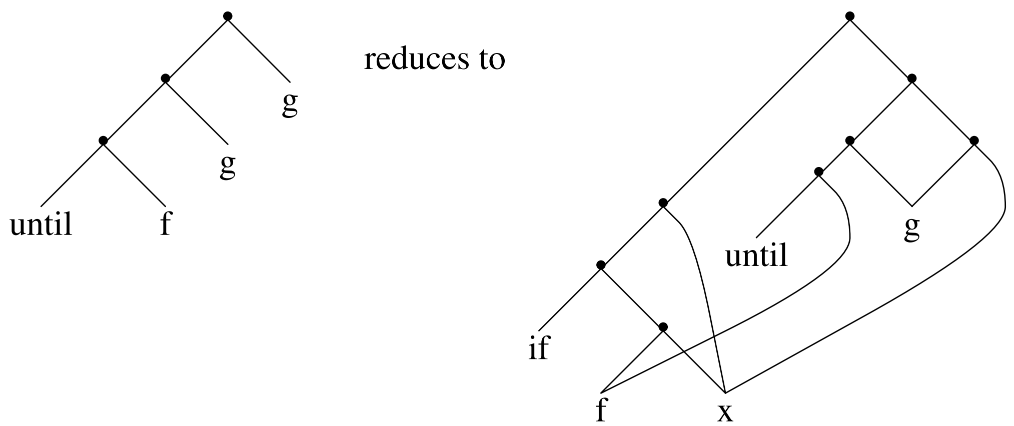
We had to fix the method by which a Ginger program would be evaluated. We had a number of choices available to us, ranging from a SECD-like implementation to a combinator or supercombinator reducer, or even fully compiled code for our Suns. The latter was ruled out on the grounds of portability, and we felt that a modified lambda-calculus reduction engine would be significantly more complex in a parallel environment for our purposes. So our choice was a combinator reducer.

The next question to ask was "which combinators?". Initially we implemented the combinator set {`S`, `K`, `I`, `B`, `C`, `S′`, `B′`, `C′`, `Y`} together with a small set of predefined operators (such as `+`, `cons` etc.) to act on the basic data types. To ensure as much flexibility as possible we were reluctant to use supercombinators, although these can be coded in at a later date if required.

Since then we have implemented as predefined operators many of the common functions (such as `map`, `foldl`, etc.) that might appear in the "standard environment" of a functional language. This has reduced the "grain" of the operators substantially, and speeded the implementation up. All the predefined operators have "complexity" of the graph rewrite at worst similar to that for the combinator `S′`. For instance, the predefined function `until` could be defined in Ginger by

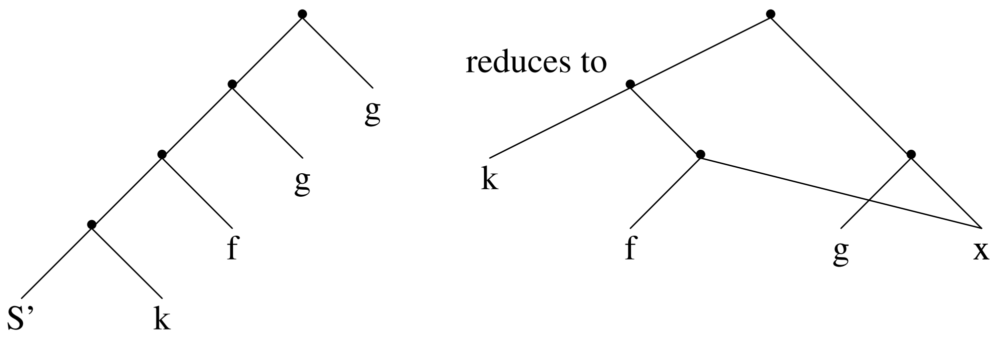
```
until f g x =
    if f x then x
    else until f g (g x)
    endif;
```

and has its graph rewrite rule



Compare this with the combinator `S′`, which could be defined by

```
S' k f g x = k (f x) (g x);
```



A side-effect of implementing so many predefined functions is that it becomes much easier for the user to visualise how a program is evaluating.

A Ginger program, when read into the machine, is initially translated to a graph representing that program as a lambda-expression. This is then compiled (using the "standard" algorithm [5]) to an acyclic graph containing extra combinators but no lambdas or names. This graph is then reduced, as described below. The graph *always* stays acyclic, and we use a "reference count" algorithm for managing storage. This has the useful side-effect that at any point during the evaluation of a program one knows the size of the graph currently in memory.

All functions are assumed to be lazy unless specified otherwise, either implicitly (such as for number operators such as +), or explicitly, by means of the operator `strict`. This operator is defined by

```
strict f x = f x;
```

but forcing `x` to be evaluated and not totally undefined before being passed to `f`.

3. The Evaluation Mechanism

Reduction of the graph takes place by rewriting several parts of it concurrently. The usual lazy evaluation strategy is embedded within this parallel reduction strategy, in the sense that at any stage in the reduction process the next reduction step in the lazy evaluation strategy will always be one of the several concurrent reduction steps in our parallel strategies.

The simulated machine has a fixed number of *processors*, together with a variable number of *processes*. Each process rewrites a section of graph until one of the following conditions is reached:

- that section becomes a datum (integer, floating point number, boolean or list);
- that section cannot evaluate further (when an operator is presented with too few arguments);
- that process is destroyed by another process.

A specific process will not recursively evaluate elements of a list, but will spawn further processes if required. A process contains (inter alia) two *stacks*, a *status* value and a *priority* value.

Each processor is either allocated a single process and is *Busy*, or is awaiting a process and is *Free*. A process may have status *Running* if in the process of rewriting a section of graph, *Suspended* if awaiting the result of another process before evaluation may proceed, or *Waiting* if it is able to continue running but has not been allocated to a processor.

A process is allocated a "priority", which is either *Possible* or *Needed*. A *Needed* process is one which is known to be required to be evaluated by virtue of lazy evaluation. A *Possible* process is one which *may* require evaluation.

Two queues of processes are maintained on each processor; one the *Waiting* processes and the other the *Suspended* processes. The elements of each queue are ordered by process priority.

- Each queue has its elements ordered by *Priority*, with *Needed* processes at the head of the queue.
- When a process in the *Running* queue completes execution, that process ceases to exist and the processor allocated to it becomes *Free*.
- If a process becomes *Suspended*, it is placed in the *Suspended* queue and the processor allocated to it becomes *Free*.
- If data becomes available to a *Suspended* process causing it to be able to continue evaluating, it gains status *Waiting* and is moved to the *Waiting* queue.
- A processor which is *Free* is allocated to the process at the head of the *Waiting* queue. If the *Waiting* queue is empty that processor simply remains *Free*.
- If a processor is *Busy*, and its *Running* process has priority *Possible*, and one of its *Waiting* processes has priority *Needed* (which we may assume to be at the head of the *Waiting* queue), then that *Possible* process is swapped with the head of the *Waiting* queue.

Time is divided into *cycles*, and in each cycle a processor may perform at most one graph rewrite. At the end of each cycle, housework is performed on the queues to ensure that, when a process has its status (or priority) changed by another process, that its correct position in the correct queue is maintained. At this point, the simulator can introduce statistical information related to the costs associated with such interprocess communication.

At the start of a program execution, after the combinator graph has been created, a single process is created with priority *Needed*, which will begin to rewrite the graph using the standard lazy algorithm. Where necessary it will create other processes with priority *Needed*, or indicate to the master process that other processes with priority *Possible* should have priority updated to *Needed*. Further processes may be created as dictated by the parallel evaluation strategy.

4. Parallel Constructs in Ginger

When modelling a machine with *n* processors, the processors are considered to be numbered *0* through *n-1*. The function

```
parallel m x y
```

returns `y`, but causes expression `x` to be evaluated on processor `m`. When a shared-memory machine is modelled, this simply causes a new process to be created on processor `m`. On a distributed memory machine, when the evaluation of `x` is complete, its value is left on processor `m`, and migrated to its parent processor upon demand from that processor.

Ginger also contains primitive functions which define the processor topology of the architecture being simulated. These are

`leftchild`: `leftchild n` returns the processor which is the left child of processor `n` in a binary tree network. If the real network is not a binary tree, then a binary tree network is mapped onto the actual network, and appropriate results are given as if the network were a binary tree.

`neighbours`: returns a list of the processor numbers which are the nearest neighbours of its argument.

`rightchild`: `rightchild n` returns the processor which is the right child of processor `n` in a binary tree network, as for `leftchild`.

`whereis`: returns the number of the processor on which the graph representing its argument is located.

5. Parallel Data Structures

The primary data structure in Ginger, as in most popular functional languages, is the list. Ginger has conventional lists based on the usual primitive functions: `head`, `tail`, `cons`, etc. Lists defined in this way are difficult to implement and to use efficiently in parallel programs, however. Recently we have shown that an alternative set of list primitives [1,2] is much better suited to parallel programming. These new lists are included in Ginger in addition to traditional head-tail-cons lists.

The new model of lists has the following six primitives:

```
[]
```

is the empty list.

`singleton x` (or `[x]`)

is the list which contains a single element, `x`.

`concatenate s t` (or `s++t`)

is the list formed by concatenating the lists `s` and `t`.

```
split s
```

is a pair of lists got by partitioning the list `s` into two parts. It is defined only if s contains at least two elements. Both lists are non-empty. If `s` contains more than two elements, the result of applying split is non-deterministic, i.e. there is more than one acceptable solution and an implementation is free to choose which of these to give as the result.

`length s` (or `#s`)

is the number of elements in the list `s`.

```
element s
```

is the only element present in the singleton list s. This function is undefined for lists which contain either more or less than one element.

The primitive `split` is defined to be non-deterministic. The implementation in Ginger happens to be deterministic, but the programmer should not make use of this fact.

The result of applying split is a pair of lists (not to be confused with a list of two lists). Now Ginger does not allow the definition of pairs of objects in a single statement, such as:

```
(s,t) = split u
```

Instead, a semantically equivalent approach is adopted and we define two primitive functions, `split1` and `split2`:

```
split u = (split1 u, split2 u),
```

so that the definition of s and t can now be carried out separately:

```
s = split1 u
t = split2 u
```

Algebraic Specification

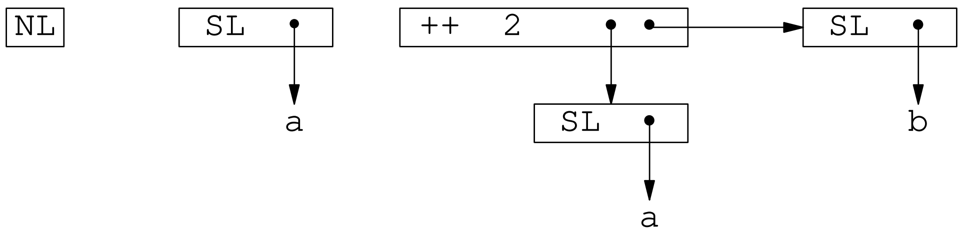
The algebraic properties of the primitive functions that can be used to specify the semantics of the new model of lists are as follows:

```
#[] = 0
#[x] = 1
#(s++t) = #s + #t
element [x] = x
s++[] = []++s = s
s++(t++u) = (s++t)++u
split([x]++[y]) = ([x],[y])
#u >= 2,  split u = (s,t) IMPLIES
    s++t = u,  #s >= 1,  #t >= 1
```

Representation

These new lists, which we call divide-and-conquer lists (or DC lists, for short) are represented as a binary trees. Each node in the tree is either a branch node or a leaf node. Each leaf node contains an element of the list. Each branch node contains two pointers, the left one points to the first part of the list, while the right points to the second part of the list. Ideally these two parts of the list should be approximately equal in length (i.e. the tree should be be balanced), but that is not a requirement for correctness of the representation. It affects only the performance. Each branch node also contains the number of items in its sub-tree (i.e. the length of the list which that sub-tree represents). Again, this is solely to improve performance (so that the length primitive can be computed in constant time).

The representations of the empty list `[]`, a singleton list `[a]`, and a list of two elements `[a,b]` are, respectively:



`NL` is the type of a node representing an empty list, `SL` is the type of a singleton list, and ++ is the type of a list of more than one element.

Lists of more than two elements can be represented in more than one way, depending on how they were generated (the representation of `a++(b++c)` will differ from that of `(a++b)++c`, although they are identical lists). The more elements the list contains, the more different tree structures can be used to represent it. All are equally valid and will give exactly the same results, although the performance of a program may depend upon how well-balanced the tree is.

With the above representation, all six primitive functions are easy to implement and execute in constant time, irrespective of the lengths of the lists involved.

6. Current Status

The simulator currently works simulating a shared-memory multiprocessor. When modelling a uniprocessor its speed is comparable with currently available Miranda or Haskell interpreters.

Primitives both for "conventional" and for "divide-and-conquer" lists have been implemented, and performance for a variety of test algorithms for divide-and-conquer lists is as predicted.

Work is currently progressing porting the code to a transputer network. At the same time the simulation of a distributed processor network is being completed. These will be used to investigate various techniques for achieving parallelism in the implementation of functional programs by graph reduction. The parallel language constructs currently in Ginger are not finalised, but are still subject to change as this work progresses and more experience is gained of using them in a fully distributed environment.

The transputer-based implementation of Ginger is being done by Neville Thomas at the University of Birmingham using the Inmos ANSI C toolkit.

References

1. T.H. Axford, ‘‘The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design’’ in *Advances in Parallel Algorithms,* ed. L. Kronsjo and D. Shumsheruddin, pp. 26-65, Blackwell (1992).
2. T.H. Axford and M.S. Joy, ‘‘List Processing Primitives for Parallel Computation,’’ *Computer Languages* (1992). To appear.
3. P.R. Hudak and P. Wadler, ‘‘Report on the Functional Programming Language Haskell,’’ Research Report CSC/89/R5, Department of Computer Science, University of Glasgow, Glasgow (1989).
4. S.L. Peyton Jones and M.S. Joy, ‘‘FLIC - a Functional Language Intermediate Code,’’ Research Report 148, Department of Computer Science, University of Warwick, Coventry (1989). Revised 1990.
5. D.A. Turner, ‘‘A New Implementation Technique for Applicative Languages,’’ *Software - Practice and Experience,* 9, pp. 31-49 (1979).
6. D.A. Turner, ‘‘Miranda: A Non-Strict Functional Language with Polymorphic Types’’ in *Functional Programming Languages and Computer Architecture,* ed. J.-P. Jouannaud, pp. 1-16, Springer-Verlag, Berlin (1985). Lecture Notes in Computer Science 201; ISBN 3-540-15975-4; Proceedings of Conference at Nancy.