

Implementing ML on the Fujitsu AP1000

Peter Bailey
Malcolm Newey
Department of Computer Science
Australian National University

Abstract

The CAP ML project seeks to develop a version of ML that is suitable for use on a distributed memory multiprocessor architecture such as the Fujitsu AP1000. Language extensions are proposed that have been developed in conjunction with a programming methodology that is appropriate to that of a massively parallel computer whilst retaining a functional style. The implementation, which is based on the SML/NJ compiler and the SML2C compiler, is in progress. This paper focusses on the language design.

Introduction

When one contemplates implementing a functional language on a parallel computer, what first comes to mind is the well known advantage claimed for functional languages, that freedom from side-effects allows multiple processors to be utilised for the parallel execution of multiple arguments in any function call. There is also a long history of applicative language compilers which take advantage of the property of referential transparency, for a variety of purposes. For example, the implementation technique of graph reduction depends on it for achieving speedup through concurrency. However, experience indicates the opposite - it is *hard* to harness a multiprocessor for ML.

There are several lessons that have been learned that are important background as we embark on the enterprise of providing a functional programming capability on a machine such as the AP1000.

- Purely functional languages are not found suitable for large application programs. Lisp and ML are far more widely used than Miranda; although the latter has been widespread for some time, its niche in the market is in the education sector. Programmers still find important uses, in major applications, for global data structures that are updated, and for i/o operations that can't readily be characterised in functional terms.
- Although ML and Lisp both allow assignments to variables, the use of this feature is discouraged in situations where concurrency is expected. (ML discourages any such use by the very syntax of reference operations.) Thus the programmer should seek to structure a program so as to have as few routines as possible that side-effect the state.
- It is only sensible to spawn a process if it is likely to survive for a time which is long compared to its setup time. Experiments where a compiler does its best to recognise which sets of subexpressions can be executed concurrently show there is surprisingly little chance that substantial parallelism can be achieved in that way; some estimates suggest that typical Lisp programs would be unlikely to make effective use of just ten processors.

- The conclusion of all implementors of Lisp and ML compilers for parallel machines is that the programmer should design algorithms with concurrent execution in mind and give explicit advice about which parts of a program can be usefully mapped to processes. A common design decision is for the system to spawn processes *only* at places where the user advises that multiple arguments to a function should be evaluated concurrently.
- Most parallel Lisp and ML compilers have been implemented on shared memory multiprocessors and really depend on this fact for their success. The rule-of-thumb that has been suggested is that each process should run for some thousands of instructions to have a cost-effective existence. In many applications a programmer is likely to find many appropriate situations.
- In a distributed memory machine, each processor has its own memory and so the setup cost includes identifying (by following pointers) all relevant cells, copying these across a network, and initialising a new heap space with this data. The operation is something like a garbage collection so we would expect it to sometimes be most economical to simply copy all of the heap space from one processor to another. In the distributed memory case, spawning processes is likely to be appropriate only if such processes last for some millions of instructions, hardly a function call.

Fujitsu AP1000 Cellular Array Processor

The AP1000 is a highly-parallel *scalable* computer with distributed memory. Each cell consists of a SPARC CPU, a Weitek FPU, custom message controllers, and 16Mb of local memory. It has a front-end host Sun4/330 which connects to the cells. These in turn are connected by three separate high-speed networks - a 2D mesh-connection torus network, a broadcast network for one-to- n communications, and a synchronisation network. Typical sizes are 64, 128, 256, 512 and 1024 cells.

This new style of expensive machine is of the same class as the Thinking Machine Corporation's CM-5, Intel's iPSC and Touchstone Delta. They share the common features of difficulty of programming and paucity of software (beyond the C and Fortran compilers). The vanguard of applications are those based in the numerical analysis of scientific problems. In these applications, processors are typically allocated to independent calculations or computations on a hunk of an array. Typical techniques are Monte Carlo simulation and Finite Element methods, just as employed on SIMD machines.

Although MIMD machines exemplified by the AP1000 apparently have more scope for fast general purpose computation, we must learn how to use them less for Physics problems and more for AI (mathematics, knowledge, reasoning etc.).

Language Design

Fine grain concurrency, especially of the sort envisaged in graph reduction, is inappropriate where there is no shared memory. Because functions are closures, there will be potentially large amounts of heap space that must be copied from one processor to another, even for quite short expression evaluations. We see very coarse grain parallelism as necessary for the combination of distributed memory machine and applicative language.

It is expected that the programmer will carefully design algorithms with concurrency in mind and will take complete charge of the processes, both as syntactic objects and dynamically

executing entities. Since we insist that the extensions to ML should retain the applicative flavour of the standard language as much as possible, processes will take arguments and yield a result.

The AP1000 style of architecture imposes considerable overheads on process creation, and thus we adopt a programming methodology that discourages the use of more processes than there are processors. The programmer should aim to create the required processes early in the execution of a program and expect that they will last for a time that is long compared to the process startup time.

In order that long-running processes can be used successfully, we must allow them to cooperate by passing information. In the case of the AP1000 this can only mean we provide communication between processes by message passing or by distributed shared memory; currently, we have chosen message-passing as the most efficient system to implement.

This design of the language is intended to support a two-level style of program structure where the top level is the initiation of ‘actors’ that interact with each other by message passing. Within these top level processes, the programming should resemble that of whole ML programs, where I/O is replaced by message traffic among the ‘actors’. Use of messages is certainly not referentially transparent but the careful programmer can still structure each process in the applicative style and write most component functions to be side-effect free.

Based on these major design decisions, we present *paraML*, an extension of Standard ML that we claim is suitable for programming the highly parallel computers of the future. The AP1000 is a leading example, being a machine with many powerful processors, each with its own large local memory. The major extensions to Standard ML are given below.

Processes in ParaML

We make changes to two areas of ML to accommodate our notions of processes. We add a new sort of declaration (ie process definition) and two new forms of expression - one to create a process instance and one which gets the value computed by the process.

Process Definitions

The ML code that is the abstracted form of a process is declared in a way that is very similar to a function definition. The difference is that the external view of a process must reflect the fact that the process can receive messages on named channels. Thus the type of a process will have the form similar to $\alpha \rightarrow \beta \rightarrow \gamma$ where α is the type of the argument supplied, β is the type (a record type) of the n-tuple of channels, and γ is the type of the result of executing the process to completion. The complete syntax is given in Bailey [1] and a forthcoming manual, but the following is the usual way in which a single process form is defined.

```
declaration:
    define pdef (channel ch1, ch2, . . . chn) pat = exp;
binding:
    val pdef = prd: 'a->'b ->> 'c
```

`pdef` is bound to the process form described in the definition; it is an ML routine that takes an argument matching `pat`, that produces a result by evaluating `exp` and that receives messages on the channels `ch1`, `ch2`, etc. `pdef` is called the process definition identifier. The words `define` and `channel` are new reserved words for ML and `->>` is a new type constructor that is used in expressions for the types of process forms and process instances.

Of course, some processes that we wish to define will read no messages and so have no need of the channel list. However, this must be signified, like the unit argument to functions. The following syntax is appropriate in this case.

```
declaration:
    define pdef nochannels pat = exp;
```

The indicator, `nochannels`, is a new reserved word. Finally, there may be several process definitions written in the same process form.

```
declaration:
    define pdef (channel ch1, ch2, . . . chn) pat1 = exp1 |
        pdef (channel ch1, ch2, . . . chn) pat2 = exp2
    and    pdef2 nochannels pat exp;
```

Process Creation

Each instance of a process is created in a *create expression*; the code associated with a process form is applied to the arguments supplied (which must, of course, be of the right type) and a running process is then in existence until the expression in the selected clause yields a result. This newly created process executes concurrently with the process containing the create expression. Although we introduce the notions with a simple, but very typical instance, the full syntax can probably be inferred.

```
create p = pdef exp1 in exp2 end;
```

The syntax is intentionally close to that of local declarations, since the scope of the process identifier, `p`, is just `exp2`. `pdef` is a process definition identifier and `exp1` is the argument that will be bound to the formal parameter of the process program. After this binding, process `p` is active and executes concurrently with the evaluation of `exp2` (called the body of the create expression). The value that results from evaluating `exp2` is deemed to be the value of this create expression.

Within its scope (`exp2` in the above example), a process identifier is taken to be of type $\alpha \rightarrow \beta$, where α is the type of the channel record and β to be the type of the result.

It is possible to initiate multiple processes in the one create expression as the following example indicates:-

```
create p1 = pdef1 exp1
and    p2 = pdef2 exp2
in exp3 end;
```

Getting Results

In a create expression, the process identifier's scope is the body (of the create expression) so that it can be used to reference the result of the process (when that result is available), to send messages to the process and as a process descriptor that can be passed to other processes.

There is a polymorphic operator called `result` which takes a process as its argument and yields the result that was produced by that process; `result(p)` will not return anything until process `p` has terminated.

Message Passing

The way to send a message to another process is by invoking the predefined function *send*:

```
send p #> c exp
```

In this example, *p* is a process identifier and *c* is one of the channel names of the process form of which *p* is a process instance. The construct *p #> c* has type α *channel* and for type consistency the message expression should have type α . The semantics of the expression is that the object that *exp* evaluates to is sent on channel *c* to process *p*. The type of *send* is α *channel* $\rightarrow \alpha \rightarrow \text{unit}$. The sending process is not blocked. The only possible exception that can be generated by a *send* is where the process *p* has terminated. If process *p* does not have a channel *c*, then the error is detected by the type checker.

The messages that are sent to a process on one of its channels are extracted from that channel (a message queue) by the predefined function *get*, an example of which follows:

```
expression:
    get #<c
result:
    x: 'a
```

In this case, *c* is a channel of the current process, the type of which must have been α *channel*. Both *#>* and *#<* are new operators, chosen to resemble the IO redirection of UNIX.

An Example

The Sieve of Eratosthenes (SOE) is a classic problem with various solutions being algorithms that are capable of efficiently using a large number of processors. In the solution below, we have a pipeline of processes, each one of which takes care of the selection of one prime number. A sequence of all odd numbers which is fed into one end of the array, is filtered as it passes along, so that the sequence that goes to the *n*th process contains no multiple of any of the first *n* primes but contains all other members of the original sequence. When the sequence is reduced to nothing a message flows back the other way, gathering primes as it goes.

```
val sieve (channel data:int) ()
= let prime = get #<data
  in if prime ==-1 then nil
    else create s = sieve ()
      in let fun f -1 = send s#>data -1
          fun f dv = if dv mod prime <> -1
                      then (send s#>data dv; f(get #<data))
                      else f(get #<data)
          in f(get #<data);
            prime::(result s)
          end
      end
  end
end;
```

The main program is the following function:

```

fun soe(0) = nil |
  soe(1) = nil |
  soe(2) = [2] |
  soe(n) = create s = sieve ()
          in let fun genlist g = if g<= n
                                then (send s#>data g; genlist(g+2))
                                else send s#>data -1
              in genlist(3);
                2::(result s)
              end
          end;
end;

```

Methodology

There are a number of standard ways of structuring parallel programs, such as worker farms, space partition, data partition etc. We have coded examples of these to show that an application that is amenable to solution by one of these strategies, can readily be written in *paraML* without the programmer worrying about process creation and inter-process communication.

There is insufficient space available to properly discuss the various recipes, so the reader is asked to watch for a subsequent paper.

Implementation

The SML of New Jersey compiler was used as the starting point for the implementation of *paraML* on the AP1000. It is incomplete at this stage although sufficient of the task is done to have uncovered some interesting problems. These problems and their solutions are presented in [1] and will also be addressed in a forthcoming report.

References

- [1] P. Bailey, “*paraML* a parallel extension of ML,” B.Sc.(Hons) Thesis, Dept. Comp. Sci, Australian National Univ., (1991).
- [2] R. H. Halstead, “MultiLisp: A Language for Concurrent Symbolic Computation,” *ACM Transactions on Programming Languages and Systems* 7, 4 (October 1985), 501–538.
- [3] M. C. Newey, “Towards a CAP Implementation of ML,” *Proceedings of 1990 CAP Workshop*, Fujitsu, Kawasaki (Nov. 1990).