

An Extension of ML for Distributed Memory Multicomputers

Peter Bailey and Malcolm Newey
peterb@cs.anu.edu.au mcn@cs.anu.edu.au
Department of Computer Science
Australian National University

Abstract

This paper describes the design of *paraML*, an extension of ML with primitives for parallelism that is suitable for use on a distributed memory multicomputer architecture such as the Fujitsu AP1000. Consideration of the lessons learned from previous parallel functional languages provided a focus for determining the requirements of parallel extensions of ML. These requirements are detailed, and the *paraML* primitives to support them are introduced. The language constructs for introducing parallelism to ML make no change to the syntax of Standard ML. These constructs have been developed in conjunction with a programming methodology that is appropriate to that of a massively parallel computer whilst retaining a functional style. An example program is used to show the succinctness of the extensions to ML provided by *paraML*. The implementation, which is based on the SML/NJ compiler and the SML2C compiler, is in progress.

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 in the attempt to achieve speedup through concurrency. However, experience indicates that there is little speedup to be gained from parallel evaluation of function arguments. In trying to harness a distributed memory multicomputer 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. “Mostly functional” languages, such as Lisp, Scheme, 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 cannot 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.) If two or more processes can update a variable, facilities for maintaining coherency must be provided. The use of shared mutable state

variables encourages an imperative style of programming, rather than a functional one. Alternatives to shared mutable state for communication and synchronisation are thus important.

- 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 should give explicit advice about which parts of a program can be usefully mapped to processes. A common design decision is to specify that the system will spawn processes *only* at places where the user advises that multiple arguments to a function may 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. There are few applications in which a compiler 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 it may 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 several thousands or tens of thousands of instructions.

The other area of interest to the development of a parallel ML is the proliferation of concurrent and distributed extensions to ML. Notable among these are Concurrent ML (CML) [8], Poly/ML [6], Distributed ML (DML) [4]. Reppy's work with CML is perhaps the most well known, and provides a semantics and methodology based on the notion of first-class synchronous events. DML builds on the basis of CML, providing asynchronous multicast operations to improve the efficiency of the extensions in a distributed environment. The primary motivation for these language extensions is to facilitate greater expressive power for operations that are most naturally expressed using concurrent threads of control, such as windowing interfaces. As such, all of these ML variants have followed a development path of being implemented on a uniprocessor, then a shared memory multiprocessor, before possibly moving to a distributed memory machine or distributed network of workstations.

While the ability to express concurrent abstractions in programming is essential, our research focuses on providing abstractions that assist in programming the massively-parallel computers of the future. To this end, the provision of abstractions that cause the explicit creation of processes, where these processes are entities which may execute in a totally new runtime system, are essential components in the design of the extensions. Similar work on a distributed version of ML is progressing at Edinburgh [7]. The development began with Poly/ML, and followed a similar path to CML. It has been recently ported to a distributed network of workstations, but processes operate with a global address space and a local address space, which distinguishes it from our developments, which are intended to execute without access to a global address space.

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 Sun4/390 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 machine is of the same class as the Thinking Machine Corporation's CM-5, Intel's iPSC and Touchstone Delta (albeit with different network topologies). 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 put more effort into learning how to use them for AI (mathematics, knowledge, reasoning etc.) and less into physical or engineering problems.

Language Design

A number of issues are involved in the design of a parallel language. In particular, designers must be concerned with granularity, implicit or explicit process invocation, communication, synchronisation, and the program model.

Coarse Granularity

Fine grain concurrency, especially of the sort envisaged in graph reduction, is quite inappropriate where there is no physical shared memory, or where the cost of non-local data access may be orders of magnitude more expensive than local data access. As functions are closures, there will be potentially large amounts of heap space that must be copied from one processor's runtime system to another. For a process that performs the evaluation of an expression, the cost of that evaluation must outweigh the initiation costs in both the creating and created processes if the process creation is to be efficient. Thus we see very coarse grain parallelism as necessary in the combination of a distributed memory machine and a "mostly functional" language.

Explicit Process Invocation

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. Process invocation will not be implicit, because this tends to result in very fine grain computation. The constructs for *paraML* should retain the applicative flavour of the standard language as much as possible, and hence 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 create the required processes, planning that they will last for a time that is long compared to the process startup time. Although the implementation may provide scheduling techniques to permit more executing processes than processors, perhaps using continuations as in CML, this form of scheduling is likely to slow down execution.

Communication by Message Passing

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.

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. Reppy comments in his discussion of CML [8] that the alternative to message-passing - mutable state - “leads to an imperative programming style,” which is not to be desired as we want to retain the applicative nature of ML. The other advantage of using message-passing is that it tends to result in greater data locality, according to research by Lin and Snyder into programming models for shared-memory multiprocessors [5]. Data locality becomes even more critical for distributed-memory multicomputers because of the non-local memory access costs.

Synchronisation Abstractions

Some form of interprocess synchronisation is useful. Various forms of synchronisation could be provided, either by new primitives, or by structuring communication primitives to achieve the same result. In CML, synchronisation is achieved directly by the extensions, which provided first-class synchronous events. In a distributed memory machine, such synchronisation would be undesirable, inefficiently synchronising all operations. In DML, the implementors have addressed the inefficiencies by providing asynchronous multicast communication. As with CML, we have provided powerful basic primitives, which may be structured and abstracted in a number of ways to support different forms of parallel programming. We believe that the given message passing primitives are adequate for synchronisation abstractions.

Program Model

The 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’. Processes may be characterised as objects that may receive messages on typed communication channels, whilst evaluating a function expression. The result of this evaluation will be available to other processes.

The use of ML modules will permit various forms of abstraction of the details involved in parallel program execution. It will be possible for a programmer to write modules that support, for example, worker farm computations or finite element computations. Having written these, another user may utilise the module without knowing anything about the details of parallelism, other than providing functions and data to execute. Different module forms may be optimised, whilst remaining transparent to the end-user.

Based on these major design decisions, we present parallel constructs for 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 operations we have provided for *paraML* are given below.

Processes in *paraML*

Parallel programming requires a minimum set of primitives to provide process creation and to enable interprocess cooperation. In *paraML* there are mechanisms for forming *process forms* (without executing them), and to create processes as instances of such process forms. A *paraML* mechanism can be used to ascertain the *result* of a process and message passing primitives are provided for communication between processes.

Process Definitions

The ML code that is the abstracted form of a process is declared in a way that is a function definition. The function must produce something of type `Process`, and this is achieved by using the *paraML* primitive `create`. The `create` operation takes an argument of type `Channels` and a unit lambda expression, which can use the formal parameter(s) to the function. A `Process` type is parameterised such that the external view of a process reflects the fact that the process can receive messages on named channels.

The type of a process closure will have the form $\alpha \rightarrow (\beta, \gamma) \text{ Process}$, where α is the type of the argument supplied, γ is the type of the channels construct, and β is the type of the result of executing the process to completion. The following is the usual way in which a single process form is defined.

```
declaration:
    fun pdef pat:'a = (create chs:'c (fn () => (exp:'b)))
binding:
    val pdef = fn : 'a -> ('b, 'c) Process
```

The function `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 `fn () => exp` and that receives messages on any of the channels specified in the channel construct, `chs`. The method by which channels are declared is discussed later. To support the notion of process definitions more clearly, syntactic sugar for the basic form given above will be provided eventually, in the same manner as `fun f x` is syntactic sugar for `val f = fn x`. (It involves minor modifications to the compiler's parser; in the prototype phase of implementation, we are restricting compiler modifications wherever possible.) The process definition may look like the following:

```
declaration:
    process pdef pat:'a chs:'c = exp:'b
binding:
    val pdef = fn : 'a -> ('b, 'c) Process
```

Process Creation

Each instance of a process is created simply by applying the closure which is the process *form* to arguments. The application causes the `create` function to be invoked and hence a process instance with the specified channels is fired up. As the process instance being created may be executed in another processor, the process body is wrapped up as a function; the process evaluation involves applying the *body function* to `()`. This newly created process executes concurrently with the process that created it.

```

declaration:
    val proc = pdef(exp)
binding:
    val proc : ('b, 'c) Process

```

`pdef` is the above-mentioned process form and `exp` is an appropriately typed argument that will be bound to the formal parameter of the process form function argument. After this binding, process `proc` is active and executes concurrently with the evaluation of the surrounding ML expression. The scope of the process identifier, `proc`, is governed by the normal ML scoping rules. Note that `proc` is a first-class ML value. Thus it can be used to reference the result of the process (when that result is available), to obtain channels on which to send messages to the process, and form part of other ML data structures. In particular, the process identifier can be communicated to other processes via the message passing primitives introduced later.

Getting Results

There is a polymorphic operator called `result` which takes a process identifier as its argument and yields the result that was produced by that process; `result(proc)` will not return anything (it is a blocking operation) until process `proc` has terminated. The use of `result` can ensure synchronisation of process termination where several processes are created; the creating expression ensures that the `result` operation is applied to all process identifiers created in the scope of the expression.

Channels

In order to communicate with a process instance, messages must be able to be sent and received. While the flexibility of dynamic channel creation, as in CML, is attractive, it poses difficulties for the program model that has been developed for *paraML*. In particular, since communication is asynchronous, channels on which to receive messages must be in existence as soon as a process starts to execute. The internal mechanisms that support a process's channels resemble a queue of messages, which may be added to by other processes, and removed from only by the process itself. The channels must be typed to comply with ML's typing system (and thus also eliminate operations on values of the wrong type - a common failure with messages in weakly typed languages on distributed memory machines). The channels must also be labeled in some manner, so that they can be referenced by both sending and receiving processes. To support these requirements, a channel construct is created as follows:

```

expression:
    chans {ch1=chan():'ch1t Channel,
          ch2=chan():'ch2t Channel,
          ...
          chn=chan():'chnt Channel}
result:
    CHANNELS - : {ch1:'ch1t Channel, ch2:'ch2t Channel, ...
                 chn:'chnt Channel} Channels

```

The expression given above for declaring channels would replace the `chs` argument in the declaration given earlier:

```

fun pdef pat:'a = (create chs:'c (fn () => (exp:'b)))

```

The functions, `chans` and `chan`, are primitives to construct objects of type `Channels` and `Channel` respectively. The word `CHANNELS` is the type constructor for the datatype `Channels` to indicate that there are channels. The collection of channels is packaged together as an ML record, so that individual channels may be referenced by their label name. (Note that in the example above, the type variable `'c` for the channels cannot actually be an ML polymorphic type variable, but must be of some ground ML type. The `'c` variable is used for brevity, as are the `'chnt` variables.)

Of course, some processes that the user may wish to define will read no messages and so have no need of channels. However, this must be signified, like the unit argument to functions; the word `NOCHANNELS` is the type constructor for the datatype `Channels` to indicate there are no channels.

```

declaration:
    fun pdef pat:'a = (create NOCHANNELS (fn () => exp:'b))
binding:
    val pdef = fn : 'a -> ('b,'c) Process

```

Message Passing

Sending

The way to send a message to another process is by invoking the predefined function `send`, and providing it with a channel that it associated with the particular destination process. To do this, the channels attached to a process must be obtainable. ML provides the syntax `#fieldname record` for retrieving a particular field's value from a record. It is not possible to write an ML function that accepts a record's field name as an actual parameter to the function - the field name must be explicitly used when retrieving from a record. To retrieve the `Channels` record associated with a process, the `getchans` primitive is supplied, which accepts an argument of type `Process`.

```

let val chs = getchans proc
    in send (#chname chs) exp
end

```

In this example, `proc` is a process and `chname` is one of the channel names of the process form of which `proc` is a process instance. The type of `send` is $\alpha \text{ Channel} \rightarrow \alpha \rightarrow \text{unit}$. The sending process is not blocked. The only possible exception that can be generated by a `send` is where the process `proc` has terminated. If process `proc` does not have a channel identified by `chname`, then the error is detected by the type checker. It is of course possible for a process to send a message to itself.

Receiving

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`. For a process to identify its own channels, the operation `selfchans` is provided. Since it is often useful for a process to be able to send its own process identifier to other processes, the operation `selfid` is also provided. Hence, `selfchans` is equivalent to `getchans(selfid())`.

```

expression:
    get (#cname selfchans())
result:
    x: 'a

```

In this case, the expression `#cname selfchans()` evaluates to a channel of the current process, the type of which must have been `'a Channel`. Since it is impossible to typecheck that a given channel actually belongs to the process passing it to the `get` operation, passing another process's channel to a `get` function results in a runtime exception.

Other Facilities

Other operations provided include facilities to:

1. get a value sent by a particular process;
2. get the latest value received on a particular channel;
3. check for a message to be available;
4. check for a message from a particular process to be available.

Operations to *select* from a list of (differently typed) channels and execute a corresponding function, and to *await* arrival of a message on any of a number of (differently typed) channels, are provided. These utilise the previous functions which determine if a channel is ready to be read.

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 pipeline, 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, each member of the pipeline appends its prime to the list of primes produced as a result of successor pipeline members, and yields this as its own result. The list of primes flows back along the pipeline, gathering primes as it goes.

```

fun sieve () = (create (chans {data=chan():int Channel})
  (fn () =>
    let val mydata = #data (selfchans())
        val prime = get mydata
    in if prime = ~1 then nil
      else
        let val s = sieve()
        in
          let val sdata = #data (getchans s)
              fun f ~1 = send sdata ~1
                | f dv = if dv mod prime <> ~1
                        then (send sdata dv; f (get mydata))
                        else f(get mydata)
          in f prime
        end
      end
    end)

```

```

        in ( f(get mydata);
            prime::(result s) )
        end
    end
end))

```

The main program is the following function:

```

fun soe(0) = nil
  | soe(1) = nil
  | soe(n) =
    let val s = sieve()
      in let val sdata = #data (getchans s)
          fun genlist g = if g<= n
                          then (send sdata g; genlist(g+2))
                          else send sdata ~1
          in ( genlist(3);
              2::(result s) )
          end
        end
    end

```

Implementation

The SML of New Jersey compiler was used as the starting point for the implementation of *paraML* on the AP1000. The prototype is progressing rapidly, and at this stage sufficient of the task is done to have uncovered some interesting problems. These problems and their solutions, as well as a more extensive discussion of the implementation strategy, are presented in [1,2].

Results of the prototype implementation reveal that the lower bound on the cost of dynamic remote process creation is approximately 20-25 milliseconds. The dominant cost involved is the copying of a process closure into a contiguous byte array for transmission on the message-passing network. Even for textually small process definitions, the size of the resulting byte array may be of the order of several kilobytes. One optimisation is to identify common parts of the runtime system resident on all the cells. These routines need not be copied as part of a process's closure when it is being created; on receipt of a process closure to execute, references to runtime system routines can be mapped to the local addresses of the same routines. Another optimisation is to create multiple instances of processes using a single broadcast operation; broadcasting of a single process closure will restrict blowout in process creation times for large systems.

Conclusions and Future Work

The design of *paraML*, an extension of ML with primitives for introducing parallelism that is suitable for use on a distributed memory multicomputer architecture such as the Fujitsu AP1000, has been presented. Requirements for parallel extensions for ML were motivated by consideration of previous parallel functional languages and the characteristics of distributed memory multicomputers. The *paraML* primitives to support them were introduced, and these language constructs make no change to the syntax of Standard ML. These constructs have been developed in conjunction with a programming methodology that is appropriate to that of a massively parallel computer whilst retaining a functional style.

Future work with *paraML* involves the completion of the implementation, design the inter-process exception handling mechanism, adding CML primitives for intraprocess concurrency, formal semantics for the extensions, and developing several standard parallel forms (such as worker farms) as polymorphic ML modules.

References

- [1] P. R. Bailey, “*paraML*: a parallel extension of ML,” B.Sc.(Hons) Thesis, Dept. Comp. Sci, Australian National Univ., (1991).
- [2] P. R. Bailey and M. C. Newey, “The Implementation of ML for Distributed Memory Multiprocessors,” to be published in *SIGPLAN Newsletter - dedicated issue on the proceedings of the Boulder Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, (October 1992).
- [3] R. H. Halstead, “MultiLisp: A Language for Concurrent Symbolic Computation,” *ACM Transactions on Programming Languages and Systems* 7, 4 (October 1985), pp. 501–538.
- [4] C.D. Krumvieda “DML: Packaging High-Level Distributed Abstractions in SML,” in *Proceedings of the Third International Workshop on Standard ML*, Dept. Comp. Sci., Carnegie Mellon University, (September 1991).
- [5] C. Lin and L. Snyder, “A Comparison of Programming Models for Shared Memory Multiprocessors,” *Proceedings of the International Conference on Parallel Programming* vol. 2 (1990), pp. 163–170.
- [6] D.C.J. Matthews, “Processes for Poly and ML,” in *Papers on Poly/ML*, Technical Report 161, University of Cambridge, (February 1989).
- [7] D.C.J. Matthews, “A Distributed Concurrent Implementation of Standard ML,” Technical Report ECS-LFCS-91-174, Dept. Comp. Sci., University of Edinburgh, (August 1991).
- [8] J. H. Reppy, “High-Order Concurrency,” *Ph.D Thesis*, Dept. Comp. Sci., Cornell University, (June 1992).