A Distributed Concurrent Implementation of Standard ML

LFCS Report ECS-LFCS-91-174

David C.J. Matthews

Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh
dcjm@lfcs.ed.ac.uk

Abstract

Standard ML is a functional programming language used extensively in universities and increasingly in industry. This paper discusses a concurrency mechanism which has been implemented in the Poly/ML implementation of Standard ML and has been used on uniprocessors and shared memory multiprocessors. It is now being implemented on a distributed network of Unix workstations. Each of these implementations is described.

The aim of this work is to produce a distributed system that will allow large ML programs to be run on a network of processors. Although eventually such a network might be a closely-coupled network of transputers, the initial design is intended for the sort of system that many organisations have, namely personal workstations on a local network. Making use of these out of office hours will provide a substantial improvement in the computing power available.

1. Standard ML

The Standard ML language [Mil90] was developed over the last few years as an attempt to bring together a number of strands of work on functional languages. It culminated in a language definition in which the semantics of the language are defined formally, probably the first formal definition of a full-scale language. Having a formal definition gives both implementors and users a degree of certainty that is lacking in other languages.

Standard ML has a static polymorphic type system based on type inference by unification. What this means in practice is that there is no need to put explicit type information into the program but that the compiler will automatically compute the most general type for each declaration. If it cannot find a suitable type the declaration must contain type errors and is rejected. This makes it particularly suitable for interactive program development. For programming in the large there is a module system that allows programs to be split up and developed and tested separately.

Standard ML is a functional language in the sense that functions, or more accurately closures because the environment of the function must be treated as part of it, are first-class objects. They can be passed as arguments to other functions or returned as results. The language includes updatable variables or *references*, which can contain functions as well as simpler values such as integers.

Standard ML is being increasingly used both in the academic community and now for industrial purposes. There are a number of high quality implementations, including the Poly/ML implementation used for the work described in the rest of this paper. It is used as the initial teaching language at a number of universities and is used as the main language of implementation at LFCS in Edinburgh.

2. Poly/ML

Poly/ML [Mat89] is an implementation of Standard ML which is currently commercially supplied and supported by Abstract Hardware Ltd. It implements the Standard ML language as closely as possible but also includes a number of extra features.

The Poly/ML implementation itself is beyond the scope of this paper, but several features of it are relevant to the provision of concurrency. As well as the concurrency primitives described in this paper and an X-windows package built using them, there is a persistent store system. The persistent store gives the ML program access to large amounts of data which are brought into store as required.

Efficient management of store is important for good performance and the design of the garbage collector can have a critical effect. All ML objects, including stacks and executable code segments, are held in the heap. The normal Unix stack is used only by the run-time system, which is written in C. This contains the garbage collector, persistent store handler and the interface to the operating system. In all there is about 9000 lines of C in the system.

3. Concurrency

The Standard ML language includes all the features required for sequential programming, but one area that was not addressed in the language definition was the provision of any mechanism for concurrency, sometimes called parallelism.

Concurrency is needed for two quite different purposes. On the one hand there are some applications for which the non-determinacy implicit in concurrency is an integral part. A window system must be able to respond to a user pointing anywhere on the screen and clicking the mouse button, and one of the easiest ways of programming this is to have separate processes manage events associated with each window. On the other hand there has been the increasing use of multiprocessors to speed up applications but to use this a mechanism must be provided to enable programs to be split into parts that can be run in parallel.

So far work on concurrency in ML has focussed on the first of these requirements. The Dialog interface to the Lambda theorem prover, for example, has been implemented using the Poly/ML concurrency primitives. This is a graphical front end which must be able to respond to user actions while the theorem prover is still computing. Without concurrency it would be necessary for the theorem prover to periodically check to see if the user had typed or clicked. With concurrency the theorem prover can be written without having to be concerned with events, which are only of relevance to the front-end, and yet they can all run as part of the same ML session.

The second reason for having concurrency, the speed-up to be achieved with multiprocessors, has not had so much attention, yet this area may in the long term prove to be the more profitable. This is the area currently being addressed.

4. The Poly/ML Concurrency Primitives

Before describing the various implementations it is useful to summarise the primitives themselves, and some of the reasoning behind the choice. The choice of primitives was governed by a number of factors. First of all was the need to avoid any change which would render existing Standard ML programs unusable. This precludes the addition of new syntax since existing programs which might have already used the reserved words as identifiers would no longer compile. It also requires that the new operations not allow the type system to be broken.

The obvious way of defining the primitives was as built-in functions. Their types have to be carefully chosen but, by using functions they are not being given the sort of special status that, for example, Ada tasking primitive [Bur87] have. It is perfectly possible to experiment with alternative sets of primitives written in terms of those provided, and these will have the same status within the language as the built-in primitives. Should a derived form prove to be generally useful it could be built-in for efficiency. An example of this in the language already are the *array* operations. Arrays are not actually defined in Standard ML but can be declared using lists. This would be hopelessly inefficient for many purposes so most implementations of ML contain arrays operations as built-in functions. They are however equivalent in behaviour, though not in speed, to a reference implementation in terms of lists.

The choice was made largely on the basis of an expectation of what would be relatively simple to provide semantics for, rather than on the grounds of what might be efficiently implemented. Providing a semantics for a language without concurrency is a major piece of work. Adding the non-determinacy inherent in concurrency makes the task extremely difficult. Nevertheless a semantics has been produced [Ber91] for a simple language using a variant of the primitives present in Poly/ML.

Processes are created using the *fork* function, which takes as its argument a function and runs it as a separate process. When the function returns the process dies. There is a related function, *choice* which provides mutual exclusion between a pair of processes. Of the two processes it creates, only one is allowed to do a communication and the other one will be killed.

The communications system chosen was essentially that of CSP [Hoa85]. A process sends a value on a channel to another process which receives it. Both processes are blocked until the value is passed which happens atomically. The processes are then allowed to proceed. A non-blocking version of *send* could be implemented using the blocking primitives provided, by associating a buffer with a channel. In a distributed system there might well be efficiency gains if this was built in.

Channels are typed, that is the type system ensures that the values sent and received on a particular channel have the same type. When a reference or a channel are passed through a channel they are shared between the sender and the receiver. This allows, for example, a server process to receive a request from a client which includes a channel on which to reply.

The primitive functions and their types are shown in Figure 1.

```
val fork: (unit -> unit) -> unit
val choice: (unit -> unit) * (unit -> unit) -> unit
type 'a channel
val channel: unit -> '_a channel
val send: 'a * 'a channel -> unit
val receive: 'a channel -> 'a
```

Figure1. The Poly/ML Concurrency Primitives.

5. The Uniprocessor Implementation

The first implementation of the primitives was on a Sun3 to support multiple threaded windows. On a uniprocessor there is no advantage in speed of using concurrency, although for illustration a few example programs were written. The concurrency primitives in ML were mapped directly onto calls into the run-time system, written in C.

5.1 Process Creation

Each process runs on a separate stack, with each stack an individual heap object. In keeping with the idea of processes being light weight, the stacks are initially small, but when a process overflows the end of a stack all the stack values are copied into a new larger stack segment. Other schemes, such as allocating individual stack frames on the heap, were considered but put too high an overhead on the garbage collector. Whenever a process makes a call into the run-time system, whether concerned directly with the process mechanism or not, all the registers are saved in the stack segment. The ML processes can be interrupted resulting into traps into the run-time system, but the run-time system itself cannot be interrupted.

As well as a stack segment, each process has a process base which points to the stack segment, and is used for synchronisation. The process bases for runnable processes are linked together in a chain, and time slicing involves simply moving a pointer round the chain to select the next process to run. A periodic interrupt is used to provide the time-slicing. There is no priority scheduling, although it would be possible to use some algorithm based on, say, whether a process had used all of its time-slice or whether it had become blocked for a communication on a channel or for external input or output.

Fork creates a new stack segment and process base for the function. The stack is initialised to start executing the function and to call a kill-process function in the run-time system when it returns. The kill-process function is also called by an exception handler if an uncaught exception is raised in the process. A newly created process base is added on to the run queue and so will be run when its turn comes.

Choice is implemented using similar code to *fork* for each of two processes, but a state variable is shared between the two processes created. The state variable is pointed at by the process bases of the two processes, and examined when one of the processes attempts to do a communication. If the state variable is already set the communication does not happen and the process is killed, but if the variable was clear it is now set and the communication proceeds.

5.2 Communication

A channel is a two word updatable object and the *channel* function merely allocates store for it. Each word is used as the head of a list of processes blocked on the channel waiting to do a send or a receive. Normally only one of the lists will be non-empty, but it is possible for processes to be waiting to both receive and send on a channel if they are alternative choices. When a process attempts to send or receive on a channel which cannot immediately satisfy the request, its process base is removed from the run queue and linked onto the appropriate chain. Blocked processes use no resources other than the store required for their process bases and stacks, and this store will be garbage collected if the channel is not reachable from a runnable process and so can never be woken up.

Transferring the value is the easiest part of the communication. Since both processes are running within the same memory only one word needs to be copied, and this is held in the sender's process base until the communication is complete.

5.3 Input/Output

External input and output, such as to files or to the console, are not dealt with in the same way as communications on channels. The main concern is to ensure that a *read* or *write* does not block since that would cause the Unix process to be suspended and so prevent any of the ML processes from running. A process that attempts to do an operation that will block is left on the run queue but set up so that it retries the operation when its time slice comes round again. The alternative, of setting the stream to generate an interrupt had unfortunate side-effects, and this scheme works satisfactorily provided the time-slices are not so long that the external events are unacceptably delayed or so short that the processor spends too much time polling devices which are not ready.

6. A Shared Memory Multiprocessor Implementation

The process primitives and the uniprocessor implementation were designed to support the window system so that there would be a satisfactory interaction with the user. Some examples of other programs were written to experiment with a process-based style of programming and also to see how well the implementation dealt with large numbers of processes, but the overhead of process creation meant that all these examples ran considerably slower than the equivalent programs without processes. On a multiprocessor it is possible to use the process mechanism to split a computation so that it will run faster, but the requirements on a process scheme designed for efficient execution on a multiprocessor might well be different to that required for the window system. The aim of the Firefly implementation was to investigate this.

The Firefly multiprocessor [Tha87] is an experimental system developed at DEC-SRC. Each Firefly consists of 4 Microvax processors together with an IO processor connected to a shared memory. The hardware ensures that the processor caches are consistent. The Fireflies run a kernel called Topaz which provides lightweight processes called “threads” and multiplexes these onto the available processors. Getting true parallelism in ML was a matter of using these threads to run ML processes. Provided threads are not used the Firefly can be run exactly like a uniprocessor Vax with Topaz providing Unix emulation.

Poly/ML had originally been written for the Vax processor and so porting the sequential part of the system was simple, requiring only a few hours work. The major problem was converting the run-time system to run in a parallel environment since it was not re-entrant. In the end the solution adopted was to treat the whole run-time system as a monitor and use a single semaphore for the whole run-time system. A more careful design with monitors on smaller sections of code would almost certainly have improved performance but required considerably more drastic changes.

The run-time system can be entered either by an explicit call, for example to read a character from a file, or by means of a trap. Traps occur as a result of using an object in the persistent store, resulting in an illegal address trap, or as a result of a trap instruction being executed, indicating that a heap or stack segment is exhausted.

6.1 Threads and Processes

The initial design used one thread for each ML process, but although threads are intended to be lightweight the overheads of thread creation and manipulation are still considerable. Instead the design was changed to use only four worker threads, i.e. one per processor, each forked at the start and merely stopped if there is nothing for it to do. Each thread takes a process off the run-queue and runs it until either the process blocks or its time-slice is exhausted. This is in many ways a simple extension of the uniprocessor implementation. It is obviously necessary to reduce the number of worker threads if there are not enough processes to run, though this is complicated if processes needing to do external input or output are left on the run queue, as in the uniprocessor implementation.

As well as the worker threads there are two threads concerned with trap handling. One deals with synchronous traps such as heap segments becoming exhausted or persistent store faults, the other deals with asynchronous traps such as console interrupts and time slicing. These are needed because, although Topaz emulates the basic Unix system, once multiple threads are used there are problems with signals.

6.2 Garbage collection

Garbage collection was implemented as a synchronous scheme in which all the processes were stopped, the whole store was garbage collected, and the processes were then allowed to continue. It is probable that asynchronous garbage collection would have considerably improved the performance of the system, but it would have required a drastic redesign of the garbage collector. In any case there has been a considerable amount of work in this area, including work on the Fireflies themselves [Ell88].

Each worker thread has its own segment of heap in which to allocate store. This allows the threads to run independently and avoids the need for frequent calls to a central heap allocator. When a thread exhausts its segment it traps into the run-time system and is allocated a new segment. Only when the allocator is unable to satisfy the request must the store be garbage collected.

Garbage collection requires all the worker threads to stop so that there is no activity which would affect the store. When a process traps in to the run-time system it has to acquire the run-time monitor lock, and this is used as a way of stopping activity. Each worker thread is persuaded to make a run-time system call as soon as it is in a safe state, and then it becomes blocked on the run-time system semaphore, since the semaphore is already held by the thread performing the garbage collection. As soon as all the threads have become blocked the store can be garbage collected. When garbage collection is completed the thread allocates itself the store it needs and returns from the run-time system, releasing the semaphore. This now allows one of the other threads to do its run-time system call, which then gradually releases the other threads.

7. The Development of a Distributed Implementation

Although the underlying implementation of processes on the Fireflies is substantially different from the uniprocessor version, the semantics is the same. It is important that any distributed implementation should have the same semantics so that users can move their programs from a uniprocessor to a distributed multiprocessor without seeing any difference in behaviour, except one hopes, a substantial speed-up.

The Fireflies have one serious disadvantage, namely that the individual processors were very slow, and so there is little incentive for further work on them when a four processor Firefly can be outperformed by a single uniprocessor. To continue to develop multiprocessor applications it was necessary to look for a distributed implementation.

One approach would be to design for a network of transputers, but there are a large number of personal workstations in the Laboratory which are left idle out of office hours, and it seemed sensible to design a system which would make use of them.

7.1 Data Transfer

The underlying problem with any multiprocessor system is data distribution and consistency. If programs can be split into processes that are truly independent then they can easily be run on separate processors. If they use large amounts of shared data then we have the problem of moving that data between machines. Equally if one process changes a piece of data on one machine that change must propagate to every process that has a copy. Even on the Firefly with a shared main memory this problem arises since each processor has its own cache memory and the caches have to be kept consistent by special hardware. To solve the problem on a network of Unix workstations we have to devise a scheme for data transfer.

Before describing the system in detail it is first important to distinguish two kinds of data in ML programs. The ML type system only allows values of types *ref* or *array* to be changed, objects of all other types are given their values when they are created and cannot afterwards be changed. The Poly/ML run-time system distinguishes these values at run-time. Together with the objects representing bit-maps, channels, process bases and stacks, these are collectively known as mutable objects. All other objects, including pieces of executable code, are immutable.

As well as the difference with assignment, mutability also affects the definition of equality. When two mutable objects are compared they are considered the same if and only if they are the same object, in other words they have the same address. In contrast immutable objects are equal if their values are equal. It is not possible to find out if two immutable objects are actually at the same address. An implementation is therefore at liberty to merge immutable objects with the same contents to reduce the storage requirements, should that be required, or to make multiple copies of immutable objects; the programmer will not be able to tell the difference.

Immutable objects therefore do not require any mechanism to ensure consistency, but that still leaves mutable objects. Fortunately, mutable objects are actually a very small proportion of the total number of objects used by ML programs. In C, by contrast, all objects can be changed. The data transfer mechanism for ML must be able to handle mutable objects but must be designed to be efficient for immutable objects.

The semantics of immutable objects allows the implementation to make a copy of an immutable object to send to another machine, without affecting the meaning of a program. There is however the problem of the size of a data structure. Immutable data structures could be transferred by walking over data structures, packaging them up, and sending them in single packages to the destination. This is the method used in RPC (remote procedure call) systems and is usually known as “marshalling”. There it is assumed that the data structure will be small and there is usually some implementation-imposed limit on the overall size of the package. In a language like C there is a fairly clear correspondence between data at the user-level and machine words, but in ML that is by no means true, and a user cannot be expected to know in advance whether a particular closure, say, will fit in a given packet size. A mechanism for ML must be able to deal with arbitrarily sized data structures.

7.2 The Network Address Space

To handle this we have to introduce a network-wide address space. The address space is partitioned so that each machine has part of the space. Suppose a process on one machine wants to send a value through a channel to a process on another machine. If the value is a local address it converts the address into the network form and sends that across. The network address identifies both the machine and the object on that machine. The other machine may pass the address on or may at some point send back to the originator to get a copy of the object it refers to. The originator returns a copy of the object, possibly with more network addresses in it. If we are only concerned with immutable objects, this copy is indistinguishable from the original. For efficiency a number of objects are packaged together so as to minimise the number of calls back, but this scheme allows for arbitrary amounts of data to be sent.

This mechanism also provides several alternative ways of dealing with mutable objects. A simple solution is to keep a reference on the machine on which it originates and always send a network address. The code for assignment and dereference on the receiving machine make the appropriate calls back to do the assignment or dereference on the originator. Alternative schemes could involve making copies of references but remembering where they were so that any changes could be broadcast to them [Li90]. Given that assignments are very infrequent compared with other languages, the apparent inefficiencies are not necessarily a problem. The trade-offs here depend very much on the way in which references are used.

Immutable objects include segments of machine code so it is perfectly possible for a process to send a function from one machine to another, and indeed one would expect this in a functional language. This would not make sense in a heterogeneous network, but if we confine ourselves to a homogeneous network it is perfectly satisfactory.

7.3 Implementation of the Network Address Space

Perhaps the simplest way to implement the network address space would be to include it within the virtual memory system, but one of the overriding principles was not to make changes to the Unix kernel. However, there was already the basis of a suitable mechanism in Poly/ML, the persistent store. The persistent store provides access to workspaces for users and automatically pages data in from disc. This is very similar to a network where objects have to be copied from one machine to another. Persistent addresses occupy a part of the address space which is illegal for user programs so that any attempt by the program to use a persistent address to access an object results in a signal, either a bus error or a segmentation violation. The signal is handled by a routine which brings the object in to memory, changes the persistent address into the real address of the object, and retries the trapping instruction.

This scheme can be used for network addresses. Part of the persistent address space becomes the network address space, with the network space divided between the processors. When a process on one machine tries to use a network address to access an object, it traps; but this time the trap handler makes a call to the machine encoded in the address and asks for the object, retrying the object as before.

7.4 Process Creation and Communication

The eventual aim is to implement exactly the same primitives as used on the uniprocessors and the Fireflies. However as a step in the development a separate *rfork* call has been introduced which forks a process on another processor, but is otherwise the same as *fork*. In the present implementation new processes are randomly assigned to the available processors. This apparently inefficient scheme appears to work reasonably well, provided the number of processes is large, but process allocation or even process migration depending on load and/or the demands of processes on shared data structures would certainly be worth exploring.

Process communication by channels has been implemented for the distributed system. Although channels are the means by which values are transferred between processes it is their synchronisation function which is by far the most important. Indeed the transfer of data is almost incidental. They should not be thought of as "pipes". Channel objects are never moved from the machine on which they are created, instead any communications through a channel are synchronised by the machine on which the channel object resides. As in the uniprocessor scheme a blocked process is put on a chain associated with the channel if the process and the channel are on the same machine. If they are on different machines a process wishing to do a communication sends a message to the channel and receives a reply allowing it to continue or blocking it. If it is blocked it will be sent a message in the future waking it up.

8. Development and Future Work

The basic design is intended to provide a working system on which programmers can begin to develop distributed applications. It is not particularly efficient but provides a starting point for an exploration of some of the trade-offs. Inevitably the trade-offs depend on the sorts of programs written, so it is important to encourage the use of the system in order to have realistic benchmarks. There is some degree of circularity here in that programmers will tend to use features that are already efficiently implemented. This is one reason for providing as primitives those functions which have well-defined semantics rather than those which can be efficiently implemented.

There are many issues to be explored in this area. How are network addresses garbage collected? What is the best way of dealing with references? How much data should be sent when a large data structure is sent to another process? When should processes be migrated and how should the load be balanced between the machines? While these issues have been and are being explored for other languages the special characteristics of ML make this research a particularly interesting topic.

9. Related Work

The relationship of functional languages and concurrency has been explored from a number of different perspectives and is an active area of research. PFL [Hol83] is an extension of an earlier version of ML with concurrency primitives and has been implemented on a uniprocessor. Reppy [Rep88] has provided an alternative set of concurrency primitives for Standard ML. Facile [Gia89] is a small language similar to ML but with concurrency built in. This work is particularly concerned with the semantics of concurrency and functional languages but has been implemented in Standard ML.

10. References

Ber91. David Berry, Robin Milner and David N. Turner, "A Semantics for ML Concurrency Primitives," In preparation, 1991

Bur87. Alan Burns, Andrew M. Lister, and Andrew J. Wellings, *A Review of Ada Tasking,* Lecture Notes in Computer Science, Springer-Verlag, 1987

Ell88. John R. Ellis, Kai Li, and Andrew W. Appel, "Real-time Concurrent Collection on Stock Multiprocessors," Technical Report 25, Dec Systems Research Center, 1988

Gia89. Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad, "Facile: A Symmetric Integration of Concurrent and Functional Programming," *International Journal of Parallel Programming* pp 121-160, 1989

Hoa85. C.A.R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science Prentice-Hall, 1985

Hol83. Sören Holmström, "PFL: A Functional Language for Parallel Programming and Its Implementation," *Chalmers/SERC Workshop on Declarative Programming*, University College, 1983

Li90.Kai Li, Private communication, 1990

Mat89. David C.J. Matthews, "Papers on Poly/ML," Technical Report 161, Computer Laboratory, University of Cambridge, 1989

Mil90. Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, MIT Press, 1990

Rep88. John Reppy, "Synchronous Operations as First-class Values," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* , 1988

Tha87. Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr., "Firefly: A Multiprocessor Workstation," Technical Report 23, Dec Systems Research Center, 1987