

Implementing ML on Distributed Memory Multiprocessors

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

The advent of distributed memory multicomputers, such as the Fujitsu AP1000, enables the implementation of parallel programming languages where every processing element is capable of supporting a runtime system large enough for languages such as Lisp and ML. The language ML is a “mostly functional” programming language which requires significant runtime support for features such as garbage collection. Most existing concurrent implementations of ML use shared memory and a single runtime system. In a distributed memory multicomputer, the cost of non-local memory access can be orders of magnitude more expensive than local memory access, and so these existing concurrent implementations are not suitable. Having a global address space makes porting existing concurrent ML extensions easier but the implementation may be too inefficient. Implementing ML on a distributed memory multicomputer such as the AP1000 thus requires a different approach in providing concurrency primitives. This paper describes an implementation of ML to enable the user to utilise the advantages of that style of architecture.

1 Fujitsu AP1000 Cellular Array Processor

The AP1000 is a highly-parallel *scalable* computer with distributed memory. Each cell consists of a SPARC IU, 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 Thinking Machine Corporation’s CM-5, Intel’s iPSC and Touchstone Delta (although they use different interconnection topologies).

2 Language Design

The classic problem for parallel programming languages is controlling access to memory or other shared resources by multiple processes. On a distributed memory machine, this is complicated by the non-uniform memory access costs and latency considerations. Although some distributed memory machines such as the Kendall Square architecture provide a global address space, others such as the CM5 and AP1000 do not. The costs involved in process creation on architectures such as the AP1000 and CM5 will often preclude efficient fine-grain parallelism. Previous attempts with shared memory machines have found that unrestrained fine-grain parallelism is grossly inefficient [5] and [6]. The model of threads-based concurrency for ML has also proven to be limited, according to Tolmach and Appel in [9]. Until machines like the Parallel Inference Machine which support efficient fine grain parallelism are more commonplace, we believe that a two level view of concurrency is required. We have chosen to base our design of extensions to ML on an assumption of very coarse-grain parallelism. This macro level of parallelism provides only one level of concurrency required for general purpose programming. The second level of more fine grain parallelism is not part of the design of our extensions, as we wished to provide a clean design for the macro level of parallelism. In addition, the fine grain concurrency approach is supported by the language Concurrent ML [7], and we intend to consider the possible merging of CML concurrency primitives with our extensions at a later time. A full discussion of the design of the resulting language,

paraML, is given in [2],[3], and [4].

The major decisions can be summarised as follows:

- Process forms are defined by function declarations; a process instance is created at the time an appropriate argument is applied to the process form.
- Process instances are initiated dynamically by application of an argument to a process form; their creation is like function invocation only in so far as arguments and control are passed at creation time; control continues in the parent process while a new thread of control is commenced in the created process.
- Access to the result computed by a process is by a primitive which waits for the process to complete.
- Processes interchange information with other processes by means of messages. Channels (implemented as mailboxes) are typed and are referred to by identifiers associated with particular process forms.

Although the semantics of processes was motivated by language considerations, it turns out to be simply described operationally by reference to a UNIX-based simulator for *paraML*. In this system the four decisions given above are implemented by the following (abbreviated) mechanism:

- Elaboration of a ML process form causes a UNIX fork. The child UNIX process embodies the ML process form (a closure) and simply waits for a signal indicating an instance of the ML process should be created. The parent process continues.
- When an ML process is to be created, a signal is sent to the (appropriate) waiting ‘process closure’ process which then forks (again). The new child is the created ML process instance; it reads its parameters and executes the code of the process body to completion.
- The result of computation by the ML process instance is communicated back to the creating process using standard UNIX communication primitives.
- Interprocess communications are performed using standard UNIX interprocess communication primitives also.

3 Mapping onto the AP1000

The AP1000 is an appropriate machine for supporting *paraML*. The host machine provides an interactive front-end for the user, from which to define process forms and create instances of them. Process forms are functions such that application to an argument should yield a process instance that executes on a free processing element. Thus the application of the process form to its argument is a call on a ‘create’ primitive that encapsulates the process instance as an ML object which incorporates the process’s channel description and body. To prevent its evaluation by the parent processor, the process body is communicated as a unit function. When this object is received in the destination processor the evaluation of the process body is initiated by applying the ‘body function’.

On each cell of the AP1000, a runtime system for *paraML* executes. The closure that is a process instance will be located in the cell’s heap space on arrival. As might be expected from the above UNIX implementation outline, the construction of the new process’s core image may be done in two steps. The process parameter will be bound to its value at the time the process instance is created but the rest of the environment is determined at the time the process definition is elaborated. Interprocess communication takes place by using the message-passing networks of the AP1000. What is novel about this approach is that it is possible to have a complete runtime system for ML executing on every processing element of the computer. The consequence is that there is no shared address space common to all processes. Processes may update data structures within their own environment, but to communicate information between processes requires message passing. A property of functional languages is that many of the items of data are represented by complex structures where substantial sharing of memory occurs naturally. An important area of research which arises as a result of these design decisions involves algorithms which can recover such sharing that is destroyed when complex objects, such as closures, are passed from one process to another and returned after some manipulation.

4 Optimisations

There exist many opportunities for the compiler and/or runtime system to optimise performance. For example, analysis of the program may reveal the number of process closures that will be formed, and if this is very small (as could often be the case with just 1 or 2 different types of process), the process closures may be broadcast to all cells, rather than sent individually. Other paradigms such as worker-farm and finite-element models can be provided and optimised as a set of library forms; the user can then program with these abstracted modules, rather than at the lower level of explicit process management.

5 SML/NJ, SML2C

It is a big advantage for a language if it is possible to extend the language for parallelism without any modifications to the syntax. If no syntax modifications are required, then no wholesale modifications to the compiler are necessary, and users are not required to learn new syntax in the language. Such extensions have already been done for ML on shared-memory systems with the language Concurrent ML [7]. All that is absolutely required for *paraML* is modifications to the runtime system to permit access to AP1000 operating system and library routines. The *paraML* extensions take the form of ML modules which provide a clean and concise interface, wrapping up the code which performs the *paraML* operations as simple functions.

A major advantage of using ML is that it treats functions as first-class objects. The runtime system views a function in the same manner as any item of data (such as an array or string). Structured reading and writing of any first-class object is possible, and since *paraML* processes are merely specialised forms of functions, they can be passed around between runtime systems on the cells like any normal item of data. A consequence is that dynamic process generation and typed message passing is relatively straightforward.

There are various publicly available compilers, but eventually the New Jersey Standard ML implementation [1] was chosen as it had versions for SPARC assembly code (necessary for the AP1000's cells) and there existed an SML to C compiler (SML2C) [8], based on the SML/NJ compiler, that also looked to be of use. It is advantageous to adapt SML/NJ (rather than work from scratch) since maintenance of the resulting system can rely on maintenance of the original compiler. Such an approach also takes advantage of the many person-years of development of the SML/NJ compiler.

The SML/NJ compiler produces machine code directly, but the development of the *paraML* cell runtime system using bootstrap strategies requires cross-compiling with C libraries to access cell operating system routines.

As a consequence, we worked with the SML2C compiler as it uses a library of C routines to provide the runtime system, after the SML has been compiled into C. The task of extending the library to include AP1000 primitives was straightforward. Additional changes had to be made to account for the operating system that resides in the AP1000 cells. Of particular note is the need for UNIX I/O operations, since the SML2C runtime system relies on use of such routines. The distributed file system, Acacia, developed for the AP1000 at ANU, provides support for UNIX I/O operations.

A prototype version of the *paraML* runtime systems for the host and the cells has been written in ML and compiled using the modified SML2C compiler in a bootstrap process. The fundamental primitives in the *paraML* extensions have also been completed, and made available as an ML module. Together, these provide a prototype implementation of the language.

6 Future Work

The development of various standard parallel modules for worker-farm, finite-element, Monte Carlo simulations etc., is expected to provide useful tools for users, eliminating the need for direct specification of parallelism, and hence any requirements for knowledge of the parallel extensions.

Implementation of *paraML* on other MIMD architectures such as the CM-5 is envisaged. The only modifications required are to the C primitives used for process creation and communications at the level of the ML-C interface in the runtime library.

It is clear that the initiation of processes in *paraML* will be expensive, as it is on any parallel architecture. Hence the programmer has to take responsibility for algorithm design. It will be important to

find methods to limit parallelism within the capabilities of the machine.

The SML/NJ compiler and runtime system require detailed knowledge of data objects to be encoded with the objects themselves. Such encoding is necessary for runtime garbage collection for instance. It appears feasible that creating a new class of global data objects could be done relatively easily; modifications to the compiler could map accesses to such objects so that their global location was determined, and the values accessed. Extra costs in garbage collection and coherency would also be involved, but this method would provide low-level support for a global data space.

7 Conclusion

Currently the runtime library modifications (based on the SML2C compiler) have been completed. It is possible to write ML code and compile it to run on either the AP1000's host or cells. The extensions for *paraML* have been completed as an ML module, and implemented in a prototype version of the system on the AP1000. This prototype system supports the declaration of process forms, dynamic process creation, obtaining the result of processes, and type-checked message passing on channels between processes. The use of SML2C requires that programs are compiled before execution, so interactive use of the language is not yet possible. The other restrictions at the current time are that processes may not dynamically create instances of themselves, nor can there be more processes than there are processing elements. We expect both of these restrictions to be removed by the end of the year.

We argue that *paraML* provides a clean extension to ML, without the costs of syntax modifications to the compiler, and that it is ideally suited to supporting high-level programming on distributed memory multicomputers. Its implementation on the Fujitsu AP1000 will also serve as a model for similar extensions to other "mostly functional" languages such as LISP and Scheme.

References

- [1] A. W. Appel and D. B. MacQueen "A Standard ML Compiler" in *Functional Programming Languages and Computer Architecture* pp. 301-324. Springer-Verlag, 1987. Vol. 274 *Lecture Notes in Computer Science*.
- [2] P. R. Bailey, "*paraML*: a parallel extension of ML," B.Sc.(Hons) Thesis, Dept. Comp. Sci, Australian National Univ., (1991).
- [3] P. R. Bailey and M. C. Newey, "Implementing ML on the Fujitsu AP1000," in proceedings of *ACM SIGPLAN Workshop on ML and its Applications*, (June 1992), pp. 163-168.
- [4] P. R. Bailey and M. C. Newey, "An Extension of ML for Distributed Memory Multicomputers," submitted to *Australian Computer Science Conference - 16*.
- [5] R. H. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems* 7, 4 (October 1985), pp. 501-538.
- [6] R. Goldman and R. P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *ACM Lisp and Functional Programming* (1988).
- [7] J. H. Reppy, "High-Order Concurrency," *Ph.D Thesis*, Dept. Comp. Sci., Cornell University, (June 1992).
- [8] D. Tarditi, A. Acharya, and P. Lee "No Assembly Required: Compiling Standard ML to C." Technical Report 187, CMU-CS, November 1990.
- [9] A. P. Tolmach and A. W. Appel, "Debuggable Concurrency Extensions for Standard ML," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California (May 20-21. 1991), 120-131.