

Supporting Coarse and Fine Grain Parallelism in an Extension of ML

Peter Bailey, Malcolm Newey,
David Sitsky & Robin Stanton

Department of Computer Science,
The Australian National University

Abstract

We have built an extension of Standard ML aimed at multicomputer platforms with distributed memories. The resulting language, paraML, differs from other extensions by including and differentiating both coarse-grained and fine-grained parallelism.

The basis for coarse-grained parallelism in paraML is process creation where there is no sharing of data, with communication between processes via asynchronous message passing to typed ports. Fine-grained parallelism may be introduced in a number of ways; we have investigated data parallelism and light-weight threads as per Concurrent ML. Our experiments with data parallelism and algorithmic skeletons illustrate that libraries can be constructed to hide process creation and message passing from the user. Furthermore, since processes may be arbitrarily nested, such libraries can be linked together in program hierarchies, allowing the integration of a range of parallel granularities to be used within a program in a modular and efficient manner.

Performance analysis of the system is encouraging, with good speedup for a range of problems. Optimisation of process creation and message passing have lowered the size at which paraML becomes cost-effective against sequential implementations of ML for scientific problems.

Keywords: parallel granularity, processes, ML, algorithmic skeletons, data parallelism

1 Introduction

We are interested in the implications for the use of ML and similar languages on multicomputer architectures with distributed address spaces. Standard ML[18] is a high-level programming language, which incorporates both functional and imperative programming features. The language includes advanced exception handling, strong static type checking, polymorphism, and a sophisticated module system. Prior implementations of both sequential and concurrent versions of ML have assumed a shared address space and so are not suitable for the distributed memory multicomputer systems which are typical of the next generation of computer architectures. It is important that high-level languages like ML are not discarded when applications are moved to a multicomputer system, but equally important that the extensions required to run on such a platform are efficient and well-founded. We have built an extension of Standard ML (called paraML), which satisfies these criteria, and implemented a prototype on the Fujitsu AP1000.

In a sequential ML program, there is only one environment. Similarly, Concurrent ML[21] programs are evaluated in one (shared) environment, even though there may be more than one thread of control. There are obvious differences between using

concurrent threads of control in a program in a shared environment and supporting parallel threads of control in a distributed environment. The extensions required for the latter are too heavy-weight for fine-grained parallelism, and are not aimed at a shared environment programming model. Our extensions to ML introduce new semantic objects (processes and ports), new values (process names and port names), and operations to create and manipulate them. Communication is achieved through asynchronous message passing to ports, and is non-deterministic. Earlier reports on the language definition and implementation strategy may be found in [1, 2]; this paper does not introduce or justify notation, but rather extends the earlier language work in various directions, particularly with respect to the mechanisms and motivation for supporting fine-grained parallelism.

Pragmatically, the non-uniform memory access of distributed memory hierarchies encourages the development of multiple granularities of parallelism. There has therefore been increasing interest in systems which support multiple styles of parallelism[7, 23]. The issue of whether granularity can be factored out of the development of parallel extensions remains open. However, it is also clear that some problems are best solved using a range of parallel granularities. For example, a client-server problem, in which there exist multiple clients, and a distributed server, is most likely to be expressed with processes managing the server distribution, but fine-grain threads of control within each server process to avoid problems of deadlock. Strategies to address granularity issues involve deducing fine-grained parallelism through implementation pragmas, or pushing it back to the algorithm definition level. Despite the result that more primitives are required to distinguish the two levels of parallelism, we argue that the extensions provided in paraML for coarse-grained parallelism are both necessary and different from extensions provided for fine-grained parallelism. From this perspective, our extensions can be viewed as disambiguating coarse- and fine-grained parallelism so that different levels of granularity may explicitly coexist within a paraML program.

Section 2 of this paper outlines the programming model; Section 3 introduces the mechanisms used to introduce coarse-grained parallelism. Section 4 looks at the mechanisms for introducing fine-grained parallelism, focusing particularly on the integration of Concurrent ML primitives within processes, and the implementation of a set of data parallel operations. Section 5 examines the use of library modules for algorithmic skeletons and data parallel operations. Section 6 gives some performance results for a small selection of problems, and Section 7 indicates some related work.

2 Programming Model - Abstract Machine

The discussion of the programming model that underpins paraML is an informal description of the abstract machine model for the programmer.

- A paraML program executes on a virtual machine consisting of arbitrarily many virtual processors; each physical processor has its own paraML runtime system capable of supporting multiple virtual processors. Until processes are created, the program behaves like a sequential ML program, executing on a single processor.
- Each virtual processor manages its own independent environment. When a new process is created, it is assigned to a new virtual processor.

- Processes are created by a function call; the value returned is a name that uniquely identifies the new process. The new process is an object that generates new communication port names, and evaluates an expression.
- Communication between processes is performed by sending objects to typed ports. These ports are dynamically created by a process, are private, and are readable only by the process that owns them; they are not expressible objects, and do not exist as part of the local environment. Bindings to objects are made by reading a message into the local environment from a port.

The new *semantic objects* in a paraML program are thus *processes*, which encapsulate an expression evaluating in parallel with others, and *ports*, which queue messages destined for the owning process. The new expressible *values* that permit access to these semantic objects are *process names* and *port names*. Port names are parameterised by the strong ground type of the ports they refer to, which permits static checking of communication between processes, though the communication may be non-deterministic. The concepts of processes, ports, and process and port names are similar to the *agents*, *links*, and *names* of the π calculus[17], particularly in that processes and ports are not expressible values.

3 Coarse-Grained Parallelism

3.1 Processes

Coarse-grained parallelism is introduced through the creation of processes, which encapsulate a unit of sharing. A process is an object identified by a process name, and provides two facilities. Firstly, it provides a mechanism by which new ports owned by the process can be created, yielding an identifying port name which is unique across the system. Secondly, a process accepts a thunk (a function with the unit or null argument) for evaluation. The term process is used to refer both to the object as defined above and to the expression being evaluated by the object, unless it is important to distinguish the two and it is unclear from the context which is meant.

The essence of a port is that of a queue of messages, to which anyone who knows the port name can add an object of the appropriate type, but from which only the process which owns the port can remove an object. More details on the communication mechanisms are given in Section 3.2.

The evaluation of an expression by a process is done in parallel with all other process evaluations. New ports and new processes can be created dynamically during the evaluation, and port names and process names may be passed to other processes on appropriately-typed ports.

3.2 Communication

Communication is achieved by message passing to typed ports. Message passing is asynchronous in order to minimise blocking and latency effects. Any first-class ML object (ie. an object that is expressible) may be communicated to an appropriately-typed destination port. Message arrival order is preserved for successive communication operations. A port consists of a queue of messages, and is the only mechanism for communicating information to a process. There are three operations which manipulate queues: sending a message to a port (identified by a port name),

getting a message from a port (provided the port was created by the process), and testing whether a particular port's queue of messages contains messages.

These three operations are used to construct other forms of message passing, including synchronous message passing, output ports from a process, and group-based message passing. The strategy of building more complex communication mechanisms from the simplest is beneficial, as the derived operations can be given a semantics in terms of their translation into the more primitive operations. We are working towards a formal semantic definition of the extensions.

The characterisation of processes and their ports also simplifies the task of describing in what circumstances exceptions are generated. An exception occurs when a process tries to send a message to a port, and this port is unable to queue the message. Typically, such a situation will arise because the process has terminated. The sending operation is then deemed to have failed. In many cases, it will not be necessary to check whether an exception has been generated in the remote destination. Alternatively, exceptions are raised if a process attempts to get a message from a port or test if a port is not empty, when that port is not owned by a process. If an exception is not handled during the course of a process's expression evaluation then the runtime system on the cell will report an exception for the process, in the same way that an exception propagates to the outermost level in a sequential ML program.

Various techniques developed recently, such as Active Messages[24], can minimise access latencies, but such techniques are not sympathetic with independent garbage collection and heap management on each node. In particular, handler addresses for messages cannot be guaranteed to be in the same location on every node, and message storage cannot be pre-allocated since dynamic data structures may vary in size. The introduction of mechanisms to support multiple processes running on each node, and the provision of operations at a finer level of granularity are major components of one strategy to enable access latencies to be masked with parallel extensions to languages such as ML.

The use of message passing does not provide a shared address space, but it is both simple and efficient to implement.¹ Experiments by Lin and Snyder[15] indicate that the use of the message passing paradigm, as opposed to a shared address space paradigm, can lead to greater data locality in programs. We have provided other parallel paradigms (including worker farms and data parallelism) on top of the message passing layer in a manner that abstracts details of process management and communication away from the user.

¹ An implementation aside: in building our system, we wanted to avoid wholesale changes to the existing SML/NJ compilers we utilised. Building a shared address space would have required an ability to generate memory fetches from non-local processor memories; however, 32-bit pointers are inadequate to address the available memory on a 512-cell Fujitsu AP1000 (with 16Mb per cell). Thus we would have had to change the entire representation mechanisms for ML data objects.

4 Fine-Grained Parallelism

For some problems, it is desirable to utilise fine-grained parallelism, either in addition to or instead of coarse-grain parallel constructs. For instance, client-server applications typically structure a server as a collection of cooperating threads which manage some resource. We have investigated two mechanisms for fine-grain parallelism support - the integration of Concurrent ML primitives within paraML processes (fine-grained control parallelism), and the provision of a library of data parallel operations (fine-grained data parallelism). In both cases, the problem to be solved is the scheduling and distribution of control or data across processes.

4.1 Concurrent ML Threads

Reppy's development of Concurrent ML[21] provides a clean and elegant abstraction of fine-grained control parallelism based on a notion of higher-order synchronous events. With two alterations (to permit the system-level manipulation of thread state information to be integrated with paraML's system-level manipulation of process state information) the modules which implement CML have been compiled with the paraML runtime system module to provide fine-grained parallel constructs. Primitives from paraML are used to create processes, and to provide inter-process communications, while CML primitives are used to provide concurrency within processes.

The unit of sharing for a thread is the environment encapsulated by a process. Since threads exist within a shared address space, they cannot migrate between processes for load balancing. Such a restriction stands in contrast to the Parallel Inference Machine (from the Fifth Generation Computer project), which migrates threads between processors. Although it would be possible to implement such migration in the paraML system, it would be expensive and other approaches to load balancing, such as worker farms, may be more profitably employed in paraML programs.

4.2 Data Parallelism

In contrast with the integration of CML primitives, the provision of data parallelism has been achieved through a library of routines that support declarative-style data parallel operations over collections. Data parallelism provides a fine-grained level of parallelism, where the parallelism is effective over collections of data objects, rather than threads of control. The declarative approach taken is similar to that of Blelloch in NESL[4].

Collections in paraML are managed by groups of processes, which spread the storage requirements, and enables sharing of data where the size of the collection is larger than a single physical processor's memory. (The lack of virtual memory on most modern multicomputers is frustrating.) This system demonstrates one of the strategies which can be used to reconstruct data structure sharing in the absence of shared memory. Another advantage of employing such an approach is that the library abstracts details of the management and distribution of data. The user need not know how collections are represented, since the implementation in terms of processes can be completely hidden through module interfaces. Instead, users see operations that behave as if there was a global address space. Such information hiding also permits later optimisations that are transparent to the user provided the interface remains unchanged.

```

signature BASIC_DP = sig
  type t
  type t collection
  type int collection

  dp_generate : (int -> t) -> int -> int -> t collection
  dp_map      : (t -> t) -> t collection -> t collection
  dp_filter   : (t -> bool) -> t collection -> t collection
  dp_reduce   : (t * t -> t) -> t -> t collection -> t
  dp_scan     : (t * t -> t) -> t -> t collection -> t collection
  dp_dist     : t collection -> int collection -> t collection
end

```

Figure 1 - Signature for basic data parallel operations.

In Standard ML, the unit of module construction is the *structure*; the types, values and functions exported by structures are defined in *signatures*. *Functors*, parameterised by structures, may be used to generate new structures or functors. The library of routines provided by paraML is essentially a set of basic data parallel operations, which could be used either on their own, or through a set of higher level operations. The basic operations match those of the signature declaration given in Figure 1. The functions `dp_map` and `dp_filter` are self-explanatory; the arguments to `dp_scan` (the parallel prefix operation) and `dp_reduce` must be binary associative for parallelism to be achievable. The function `dp_generate` creates a collection; the integer arguments are the number of elements in the collection and the number of processes to be used in managing the collection. The function `dp_dist` is a generalised permutation operation, which takes an integer collection describing the permutation. Lastly, there are versions of the functions which create collections of a different type from the source collection, but these have been omitted for brevity here.

When using data parallel operations, the user defines an ML structure declaring some data type and operations over it. This structure is then used as a parameter to a functor application, generating an ML structure that declaring the standard data parallel operations for this data type. Any valid function that matches the typing constraints for the functions to be provided as arguments to these data parallel operations is admissible (eg. the parallel prefix operation `dp_scan` in Figure 1 takes a function of type $\alpha \times \alpha \rightarrow \alpha$). The arguments are passed as a message to the processes representing the data collection (which can be performed very efficiently using multicast message passing in many cases). This eliminates the need to create new processes. Clearly, while this library may not provide as efficient an implementation as a compilation system dedicated expressly to providing data parallel constructs, it illustrates that programming paradigms quite distinct from the control parallel flavour of the language can be constructed. As will be seen in Section 6.2, the implementation can be made acceptably efficient (with approximately 95% efficiency gained for collections of size equivalent to 2 million integers). Unlike other data parallel implementations such as Dataparallel C[20], data parallel collections in ML may be of elements of arbitrary size and type. For instance, an element may be a list of functions; under a data parallel `map` operation, each of these functions may be applied to some argument.

This approach - providing data distribution mechanisms at a library level - stands in contrast to that taken by HPF[10], which pushes these into the compilation/runtime system layer, with directives from the user. Runtime system-level approaches to extracting fine-grained parallelism in ML-like programming languages such as lambda tagging and heap resolution[11] and future stealing[25] could be used as an alternative to requiring the user to explicitly specify fine-grained parallel operations. The tradeoffs to be made with all these techniques basically involve the effort in developing so-called supercompiler systems as opposed to the development of efficient parallel libraries. Our choice partly reflects the limited availability of resources for such compiler development and the potential for software reuse with such library structures.

5 Libraries and Algorithmic Skeletons

The mechanisms by which coarse- and fine-grained parallelism are made available to the user in paraML deliberately expose the addition of parallelism to the language. Many users wish to avoid explicitly dealing with parallel programming constructs. ML contains a sophisticated module system that enables the creation of optimised paraML versions of various standard parallel solutions as libraries. Cole describes these as algorithmic skeletons for structuring the management of parallel computation[5]. The use of such skeletons abstracts the details of coarse-grain parallelism from the user, in the same manner as the library of data parallel operations provided a model of fine-grained data parallelism at odds with the coarse-grain nature of the language extensions. The other obvious advantage of using libraries is that large parallel programs are easiest to construct when they can be built from separately developed and testable components.

The worker farm skeleton provides similar generality to that of the library of data parallel routines described earlier. The user is responsible for defining the data type on which the workers operate, and for providing functions that: initialise the system; generate a new work item; accumulate a result returned by a worker; test for whether there is more work to do; and produce a final result having accumulated all the partial results. The operations and data type which the user provide are combined into a structure, which is used to parameterise the worker farm skeleton functor. Note that the work items do not have to be hard-coded into the definition of the worker processes; we can implement the polymorphic nature of the library trivially, since work items are just functions to be passed as messages. Hence the actual function work item that is used can itself change throughout the course of the solution, provided its type remains the same. (In general though, strongly-polymorphic messages are problematic, due to the weakly-polymorphic nature of ports. This problem is analogous to the limitations on strongly-polymorphic mutable variables in SML/NJ.)

The divide and conquer algorithmic skeleton implements a common technique for solving problems. Given a problem, determine whether it needs dividing into sub-problems. If it does not, then simply solve the problem; otherwise, split the problem into sub-problems, solve each one separately, and join the results together. The parallelism comes about through concurrent solution of sub-problems. An example problem that can be solved in this manner is a merge-sort, where the list of elements to be sorted is progressively broken into smaller and smaller sub-lists. At some stage, each sub-list is sorted, and then the resulting sorted sub-lists are merged together into a single sorted list of elements.

```
signature DIVIDE_AND_CONQUER =
sig
  type prob
  type sol
  val indivisible : prob -> boolean
  val f            : prob -> sol
  val split       : prob -> prob list
  val join        : sol list -> sol
end
```

Figure 2.1 - The signature of operations and types to be provided by the user.

```
(* The skeleton combines the given functions as follows:
*
*   do_divide_and_conquer indivisible split join f = F
*       where F P = f P, if indivisible P
*                = join (map F (split P)), otherwise
*)

functor Do_D_and_C_Fun (divide_and_conquer:DIVIDE_AND_CONQUER)
  : DO_DIVIDE_AND_CONQUER =

struct
  structure divide_and_conquer = divide_and_conquer
  open divide_and_conquer

  fun F (P:prob) = (* F P = *)
    if indivisible P (* f P, if the current problem is indivisible *)
    then f P
    else (* otherwise, split the problem *)
      let (* [This function takes a sub-problem, generates a new
          process and an output port called result, and places the
          value obtained from applying F to the sub-problem on
          this output port. The function returns the output port,
          which can then be used by a retrieve operation.] *)
          fun D_and_C (sub_P:prob) =
            let val new_proc = process ()
                val result:sol OutPortName = out_port new_proc
            in
              execute (new_proc,fn () => place result (F sub_P));
              result
            end
          in
            (* and join the results of solving the sub-problems,
              each of which is solved by a process, which places
              the computed value on an output port *)
            join (map retrieve (map D_and_C (split P)))
          end
      end
end
```

Figure 2.2 - The divide and conquer skeleton.

A solution for the divide and conquer skeleton is given in Figure 2. The user provides a structure which matches the DIVIDE_AND_CONQUER signature (Figure 2.1),

including functions that determine whether a problem is indivisible, solve a problem to provide a solution, split a problem into a list of problems, and join a list of solutions into a solution. This structure is used as a parameter to the skeleton functor (Figure 2.2), which combines the given functions and manages the necessary process creation and communication requirements. The signature definition for the skeleton functor has been omitted, but is trivial. A single function `F` is yielded, which takes a problem and returns a solution. Wherever sub-problems are generated, a set of processes are created, each of which recursively applies the function `F`, and places the result on an output port (output ports are one of the derived forms of communication primitives discussed earlier, providing a blocking output communication from a process). The expression which created the sub-problems then joins the results from each process solving a sub-problem to create a solution. The primitives from `paraML` are given in bold font, simply to indicate which operations manage the parallelism of the skeleton. The solution as expressed is very close to the original functional specification of the skeleton, which is also given.

Work has been completed on the four skeletons given by Cole[5]: divide and conquer, worker farm, iterative combination, and cluster. We are also interested in other algorithmic skeletons including scattered and regular mesh decomposition, Monte Carlo simulations, and multi-dimensioned distributed array spaces.

6 Performance

6.1 Data parallel operations

Two particularly interesting sets of results from the data parallel experiments are reported here. The first (Figure 3) measures the cost of the basic data parallel operations for an integer collection of varying sizes. The arbitrary permutation (`dp_dist`) is excluded, since the behaviour of this operation varies in needing from zero to all-to-all communication. The interesting aspect of this diagram is the relatively high overhead costs for all the operations except `dp_map`, due to multiple phases of message passing and the use of 125 processes to store the data collection. Other results indicate that the size of the collection should be used as a heuristic for determining how many processes should be used in storing the data elements; only in large collections should large numbers of processes be used. These types of results will be used in the library to optimise performance. The second set of results (Figure 4) illustrate the efficiency to be gained from using `paraML`. The `dp_scan` operation has been implemented in both `paraML` and sequential ML (compiled by the same `SML2C` compiler) over varying sizes of collections. The efficiency is computed as the ratio of the `paraML` times to the serial times divided by 125. With collections of only 62500 integers, the `paraML` implementation is achieving only about 10% efficiency, but with collections of the order of 2 million integers, the efficiency is about 93%.

6.2 Black hole images

The extensions provided by `paraML` have been tested for a non-trivial scientific application, in an investigation of the use of functional languages for scientific programming[22]. The problem is a Schwarzschild black hole imaging system, which determines how the appearance of an object is distorted when its photons are deviated near the vicinity of the black hole by using ray-tracing techniques. The Bulirsch-Stoer method is used to solve the two first-order ordinary differential equations which

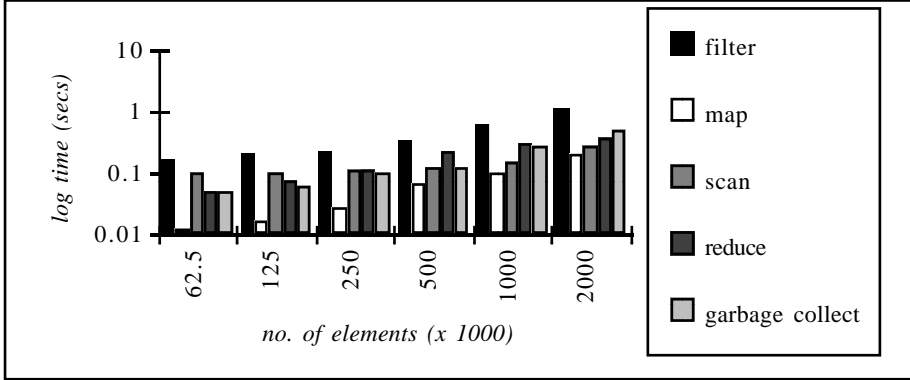


Figure 3 - Results for 125 processes - basic data parallel operations

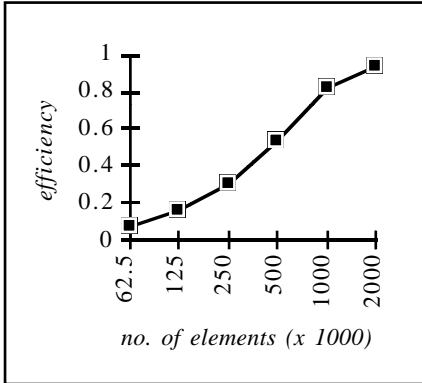


Figure 4 - Efficiency results for scan operation - paraML vs ML (SML2C) (+125)

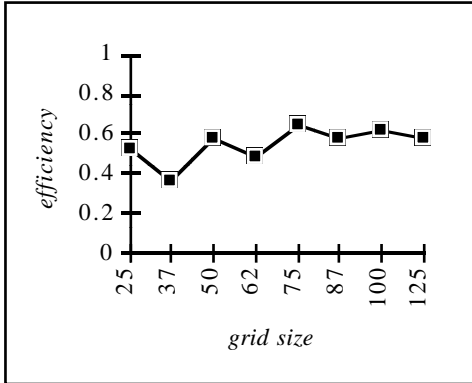


Figure 5 - Efficiency results for black hole photon trajectory imaging - paraML vs ML (SML/NJ) (+126)

describe the photon trajectories. The parallelisable component of the problem (computing the trajectories) has been coded in paraML, and the efficiency is given in Figure 5. The efficiency is calculated as the ratio of the paraML times running with 126 processes on the AP1000 to the serial times (divided by 126) of a SML/NJ implementation on a SPARC IPC (whose CPU is very close in performance to the AP1000's cells). It should be noted that SML/NJ code is significantly more efficient than code generated by the SML2C compiler for paraML, and would be the compiler of choice for high-performance scientific programming in an NJ-variant ML compiler.

7 Related Work

High-level approaches for introducing concurrency include the *futures* of Multilisp[9] and its successors such as MuT[13], and the higher-order events of Concurrent ML[21], but have been implemented under a shared address space model. At the opposite end, traditional threads packages have been used to augment ML (ML-Threads[19]), but again under a shared address space model. The language STING[12], a parallel version of Scheme, falls between CML and paraML. Other

concurrent/parallel dialects of ML include Matthews' work on Concurrent Poly/ML[16], and the work on LCS[3], which both resemble CML in being based on a shared global address space and synchronous communications. Extensions to Reppy's CML have been done by Krumvieda with Distributed ML[14], which provides more explicit support for multicast operations, and is aimed at programming distributed systems. Data parallelism under ML has been explored extensively by Hains *et al*'s Data Parallel ML[8, 6], which takes a constructive/imperative approach to data parallelism, integrated as extensions to the language definition. Their work is more mature than our brief experiments, including abstract machine and formal semantic definitions. Bletloch's work on NESL[3] has also been instructive on the requirements for effective data parallel programming.

8 Conclusion

The ability to specify a mixture of parallel granularity at a language level is novel in programming languages like ML. Various parallel extensions to ML, Scheme, and Lisp exist, but ours explicitly addresses the requirements of providing and differentiating both coarse- and fine-grained parallelism in a distributed address space. The expression of multiple parallel granularities appears to be useful for adequately programming and utilising multicomputer architectures. The language is proving to be acceptably efficient; further improvements are expected to be made, which will lower the level at which paraML becomes cost-effective against sequential versions of ML. Another significant aspect of the development is the ability to engineer modular software through composition of different program units at varying degrees of parallelism, possibly through the use of algorithmic skeletons, which provide a convenient mechanism for abstracting the details of parallel computation from the user. In order to achieve this in a succinct manner, nested parallelism, first class objects, and a good module system are required, all of which are provided by paraML.

Acknowledgments

Greg Wilson provided valuable criticism on drafts of this paper and on the data parallel implementation.

References

- [1] Bailey, P. and Newey, M. Implementing ML on Distributed Memory Multicomputers. *ACM SIGPLAN Notices*, **28** (1), 59-63 (1993).
- [2] Bailey, P. and Newey, M. An Extension of ML for Distributed Memory Multicomputers. In *Proceedings of the Sixteenth Australian Computer Science Conference*, pp. 387-396, Brisbane, Australia (1993).
- [3] Bertomieu, B. and le Sergent, T. Programming with behaviours in an ML framework: the syntax and semantics of LCS. In *Lecture Notes in Computer Science*, **788**, pp. 89-104, Springer Verlag, (1994).
- [4] Bletloch, G., Chatterjee, S., et al. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 102-111, San Diego, (1993).
- [5] Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, (1989).
- [6] Foisy, C., Vachon, J. and Hains, G. DPML: de la Sémantique à l'Implantation. *Journées Francophones des Langages Applicatifs*, (1994).

- [7] Foster, I. and Chandy, K. M. Fortran M: A Language for Modular Parallel Programming. Argonne National Labs,
- [8] Hains, G. and Foisy, C. The Data-Parallel Categorical Abstract Machine. In *Proceedings of the PARLE-93*, (1993).
- [9] Halstead, R. Multilisp: A Language for Concurrent Symbolic Computation. *ACM ToPLaS*, **7** (4), 501-538 (1985).
- [10] High Performance Fortran Forum *High Performance Fortran Language Specification*. (January 1993).
- [11] Huelsbergen, L. and Larus, J. Dynamic Program Parallelization. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 311-323, San Francisco, (1992).
- [12] Jagannathan, S. and Philbin, J. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the ACM Conferenc on Lisp and Functional Programming*, pp. 345-357, San Francisco, (1992).
- [13] Kranz, D. A. and Halstead, R. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 81-90, (1988).
- [14] Krumvieda, C. D. DML: Packaging High-Level Distributed Abstractions in SML. In *Proceedings of the Third International Workshop on Standard ML*, (1991).
- [15] Lin, C. and Snyder, L. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Programming*, pp. 163-170, (1990).
- [16] Matthews, D. A Distributed Concurrent Implementation of Standard ML. University of Edinburgh, LFCS, (August 1991).
- [17] Milner, R., Parrow, J. and Walker, D. A Calculus of Mobile Processes I & II. *Information and Computation*, **100** (1), 1-77 (1992).
- [18] Milner, R., Tofte, M. and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts (1989).
- [19] Morrisett, J. G. and Tolmach, A. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 198-207, San Diego, (1993).
- [20] Quinn, M. J. and Hatcher, P. J. Data-parallel programming on multicomputers. *IEEE Software*, **7** (5), 69-76 (1990).
- [21] Reppy, J. H. Higher-Order Concurrency. *PhD Thesis*, Cornell University (1992).
- [22] Sitsky, D. The Application of Functional Languages in Scientific Computations. *Honours Thesis*, Australian National University, (1993).
- [23] Subhlok, J., Stichnoth, J. M., et al. Exploiting Task and Data Parallelism on a Multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 13-22, San Diego, (1993).
- [24] von Eicken, T., Culler, D., et al. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256-267, (1992).
- [25] Wagner, D. B. and Calder, B. G. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 208-217, San Diego, (1993).