Converse : An Interoperable Framework for Parallel Programming*

Laxmikant V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, Joshua Yelon
Department of Computer Science, University of Illinois, Urbana, IL 61801.
{kale,milind,narain,sanjeev,jyelon}@cs.uiuc.edu

Abstract

Many different parallel languages and paradigms have been developed, each with its own advantages. To benefit from all of them, it should be possible to link together modules written in different parallel languages in a single application. Since the paradigms sometimes differ in fundamental ways, this is difficult to accomplish. This paper describes a framework, Converse, *that supports such multi-lingual interoperability. The framework is meant to be inclusive, and has been verified to support the SPMD programming style, message-driven programming, parallel object-oriented programming, and thread-based paradigms. The framework aims at extracting the essential aspects of the runtime support into a set of core components, so that language-specific code does not have to pay overhead for features that it does not need.*

1 Introduction

Research on parallel computing has produced a number of different parallel programming paradigms, architectures and algorithms. There is a wealth of parallel programming paradigms such as SPMD [14, 12], data-parallel [9, 8, 3], object-oriented [1, 3, 4, 5, 11], thread-based [7, 4, 10], macro-dataflow, functional languages, logic programming languages, and combinations of these.

However, not all parallel algorithms can be efficiently implemented using a single parallel programming paradigm. It may be desirable to write different components of an application in different languages. It will also be beneficial to combine pre-written modules from different languages into a new application. For this, we need to support interoperability among multiple paradigms. Such interoperability is not currently possible, except for a specific subset of language implementations designed for this purpose (e.g. HPF [8] and PVM [14]).

This paper describes the design and rationale of *Converse*, an interoperable framework for combining multiple languages and their runtime libraries into a single parallel program. It is based on a software architecture that allows programmers to compose multiple separately compiled modules written in different languages without losing performance. Converse also facilitates development of new languages and notations, and supports new runtime libraries for these languages. This multi-paradigm framework has been verified to support traditional message-passing systems, thread-based languages, and message-driven parallel object-oriented languages, and is designed to be suitable for a wide variety of other languages.

2 Characteristics of parallel languages

Parallel languages and their implementations differ from each other in many aspects. The next two section discuss two important aspects.

2.1 Concurrency within a process

The first aspect that is critical from the point of view of interoperability is how the language deals with concurrency within a single process (i.e. within a single processor, in most common implementations). Concurrency within a process arises when the process has a choice between more than one action at some point(s) in time. There are three categories of languages in this context:

- No concurrency/Single Threaded Modules: Some parallel programming paradigms (such as traditional message-passing) do not allow concurrency within a process. Each process has a single thread of control,

*This research was supported in part by the ARPA grant DACA 88-94-C-0019 and NSF grant ASC-93-18159.

hence a process will "block" while it is waiting for a message that has not yet arrived. During this blocking, the semantics require that no other actions should take place within the same process (i.e. there should be no side effects, when the "receive" system-call returns, beyond the expected side effect of returning the message).

- Concurrent objects: Concurrent object-oriented languages allow concurrency within a process. There may be many objects active on a process, any of which can be scheduled depending on the availability of a message corresponding to a method invocation. Such objects are called message-driven objects.
- Multithreading: Another set of languages allow concurrency by threads— they permit multiple threads of control to be active simultaneously within a process, each with its own stack and program counter. The threads execute concurrently under the control of a thread scheduler.

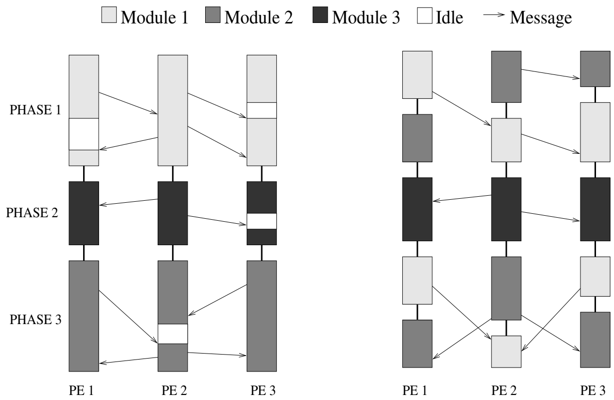
Most languages can be seen to fall within one of these three categories or combinations of them, as far as internal concurrency is concerned. Other paradigms such as data parallel languages and functional languages can be implemented in one of the above categories. For example, HPF can be implemented using a statically scheduled SPMD style or using message driven objects.

2.2 Control Regime

Another related aspect is the *control regime* for a language, which specifies how and when control transfers from one program module to another within a single process. Modules interact via *explicit* and *implicit* control regimes.

In the *explicit control regime*, (as described in Figure 1(a)), the transfer of control from module to module is explicitly coded in the application program in the form of function calls. Moreover, at a given time all processes are usually executing code in the same module. Thus all processors execute modules from different languages in a deterministic, loosely synchronous manner. This explicit control regime is sufficient for many scientific applications which can be programmed in a loosely synchronous manner, enabling different non-overlapping phases of a program to be coded in different languages. It is suitable for languages which have no concurrency within a process.

The *implicit control regime* (Figure 1(b)) is motivated by a need to reuse parallel software components in an overlapped manner, so that entities in different modules can be simultaneously active on different processes. The transfer



(a) Explicit Control Regime

(b) Implicit Control Regime

Figure 1: Control regimes for parallel programs

of control from module to module is implicit; rather than being decided only by the application program, it may be decided dynamically by a scheduling policy in the run-time system. This model allows an adaptive sequence of execution of application code with a view to providing maximal overlap of modules for reducing idle time. Thus, when a thread in one module blocks, code from another module can be executed during that otherwise idle time. The implicit control regime is suitable for languages with concurrent objects or multi-threaded languages.

3 Design and Architecture of Converse

The design of the Converse framework is based on the necessity of handling the different models of concurrency and control regimes in single-threaded modules, message-driven objects, and thread based modules. The following guidelines were used :

1. *Completeness of coverage*: the framework should be able to efficiently support most, if not all, approaches, languages and libraries for parallel programming. More concretely, any particular language or library that can be portably implemented on MIMD computers should be able to run on top of Converse, using its facilities and interoperating with other languages.
2. *Efficiency* : There should not be undue overhead for (a) remote operations such as messages, and (b) local scheduling such as the scheduling of ready threads, as compared to the cost of such operations in a native implementation.

3. *Need based cost*: The Converse framework must support a variety of features. However, each language or paradigm should incur only the cost for the features it uses.

To satisfy these requirements, the architecture of Converse is component-based, rather than monolithic. The system consists of multiple components, each of which is fully specified via a detailed interface specification. A language implementation may use only the components it needs. For each component, multiple alternative implementations may exist. Thus, an application that requires sophisticated dynamic load balancing might link in a more complex load balancing strategy with its concomitant overhead, while another application may link in a very simple and efficient load balancing strategy.

An important observation that influenced this design is the fact that threads and message-driven objects need a scheduler, and a single unified scheduler can be used to serve the needs of both. The other components of Converse are a machine interface, message managers, thread objects, and load balancers, as shown in Figure 2.

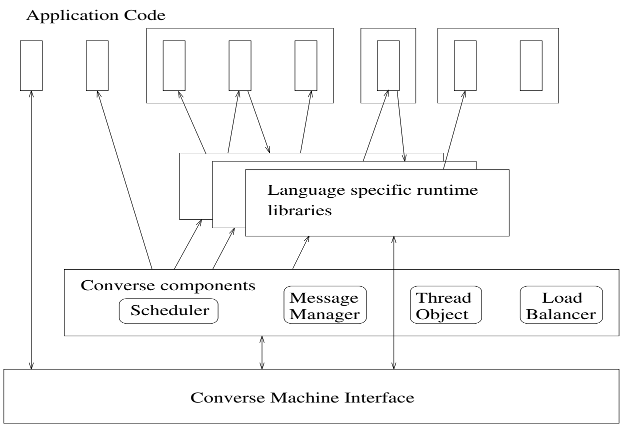


Figure 2: Software Architecture for Interoperability

When initialized, a language runtime registers one or more handlers with Converse. These language-specific handlers implement the specific actions they must take on receipt of messages from remote or local entities. The language handlers may send messages to remote handlers using the CMI, or enqueue messages in the scheduler's queue, to be delivered to local handlers in accordance with their priority.

3.1 The scheduler

The Converse scheduler is based on a notion of schedulable entities, called "generalized messages".

Generalized Messages: In order to unify the scheduling of all concurrent entities, including message-driven objects and threads, we generalize the notion of a message. A generalized message is an arbitrary block of memory, with the first few bytes specifying a function that will handle the message. The scheduler dispatches a generalized message by invoking its handler with the message pointer as a parameter. The function may be specified by a direct pointer or by an index into a table of functions. The latter method has the advantage of working even on heterogeneous machines, and requires less space than a pointer, and is therefore used in most of our implementations. Any function that is used for handling messages must first be registered with the scheduler. A generalized message may be used as a message sent from a remote processor or as a scheduler entry for a ready thread or object.

There are two kinds of messages in the system waiting to be scheduled — messages that have come from the network, and those that are locally generated. The scheduler's job is to repeatedly deliver these messages to their respective handlers. Since buffer-management issues demand timely processing of messages from the network interface, the scheduler first calls the Converse machine interface function `CmiDeliverMsgs()` for delivering network messages, which extracts as many messages as it can from the network, calling the handler for each of them. These handlers may enqueue the messages for scheduling (with an optional priority) if they desire such a functionality. After delivering messages from the network, the scheduler dequeues one message from its queue and delivers it to its handler (Figure 3). This process continues until the Converse function for terminating the scheduler is called by the user program. The scheduler's queue is implemented as a separate module so that user can plug in different queuing strategies. The handler for a particular message may be a user-written function, or a function in the runtime of a particular language.

Converse supplies two additional variants of the scheduler for flexibility. The first allows the programmer to specify the number of messages to be delivered. The second runs the scheduler loop until there are no messages left in either the network's queue or the scheduler's queue.

For modules written in the explicit control regime, control stays within the user code all the time. However, for modules in the implicit control regime, control must shift back and forth between a system scheduler and user code. For these apparently incompatible regimes to coexist, *it is necessary to expose the scheduler to the user program*,

```
Scheduler()
{ while ( not done )
  { CmiDeliverMsgs() ;
    get a message from the scheduler queue ;
    call the message's handler ; }
}
CmiDeliverMsgs()
{ while ( there are messages in the network )
  { receive a message from the network ;
    call the message's handler ; }
}
```

Figure 3: Pseudo-code for scheduler

rather than keeping it buried inside the run-time system. A single-threaded module can explicitly relinquish control to the scheduler to allow execution of multi-threaded and message-driven components. A typical interaction between the two control regimes may proceed as follows : the single-threaded module may carry out a possibly parallel computation with sends and receives, and then invoke a function f in a concurrent module (such as one written in Charm). This module may change its state and deposit some messages for other entities. When this function f returns, the single-threaded module explicitly invokes the scheduler, which executes the concurrent computations triggered by the previously deposited messages. The result of the concurrent computation is passed by function calls to the single-threaded module before the scheduler returns.

3.2 Converse Machine Interface

The Converse machine interface (CMI) contains calls which must be implemented to port Converse to any machine.

The CMI layer defines a minimal interface between the machine independent part of the runtime such as the scheduler and the machine dependent part which is different for different parallel computers. Portability layers such as PVM and MPI also provide such a portable interface. However, they represent an overkill for the requirements of Converse. For example, MPI provides a "receive" call based on context, tag and source processor. It also guarantees that messages are delivered in the sequence in which they are sent between a pair of processors. The overhead of maintaining messages indexed for such retrieval or for maintaining delivery sequence is unnecessary for many applications. The interface we propose to develop is minimal, yet it is possible to provide an efficient MPI-style retrieval on top of this interface.

The CMI module is responsible for process creation, process coordination at the initiation and termination points, communication and other machine-specific utilities. The `CmiInit` call must precede any other calls to the machine interface. The `CmiExit` call must be the last call to the CMI.

Sending messages: The CMI supports both synchronous and asynchronous send calls. `CmiSyncSend` sends a generalized message to the destination processor. When the call returns, the caller may overwrite data in message buffer. The `CmiAsyncSend` call is provided so that the application program may continue to work while the machine is trying to send message. The `CmiAsyncSend` call returns a handle that the user can probe to check the status of the send.

Broadcasting messages: The CMI provides many variants of broadcast calls, including synchronous and asynchronous ones. Note that the broadcast is called only by the processor sending the message. Thus a broadcast does not result in a barrier.

Receiving messages: For retrieving messages that have arrived on the communication network, the CMI provides the call `CmiDeliverMsgs` (Figure 3), which invokes the handler for all messages that have been received from the network. For supporting single-threaded languages which may require that no other activity takes place while the program is blocked waiting for a specific message, the CMI provides a `CmiGetSpecificMsg` call, which waits for a message for a particular handler while buffering any messages meant for other handlers.

Efficient, flexible buffer management for the received messages is an important issue. The complexity here arises due to variations in different machine and application contexts. On some machines, it may not be possible to give the user code control of the system buffer in which the message was received. Also, some application programs may be able to consume data in messages as they arrive from the network, while others may require that the message be scheduled before it is processed. To avoid buffer copying to the greatest extent possible, while still keeping the design portable, we provide the following buffer management protocol. By default, the CMI owns the message buffer. If a handler needs to retain the buffer it should explicitly call `CmiGrabBuffer(&buffer)`, which transfers the ownership of the buffer to the handler. On machines where message buffers reside in operating-system space, the CMI will transfer a copy of the buffer.

The CMI provides a number of utility calls including timers with different resolutions, atomic terminal I/O, and calls to determine the logical processor number and the total number of processors.

3.3 Message Manager

A message manager is simply a container for storing messages. It stores a subset of messages that are yet to be processed, serving as an indexed mailbox. A message manager provides calls to insert and retrieve messages. Messages may be retrieved based on one or more "identification marks" on the message. A tag and a source processor number are examples of such identification marks. Instances of message managers provided in Converse can be customized to either one or two tags. The message manager provided in Converse also allows one to *probe* for the existence of a particular message specified by its tags. A "wildcard" may be specified in the tag field. The message manager can be used by multi-threaded as well as single-threaded modules.

3.4 Threads

In many parallel programs each process has a single thread of control : they have a single stack and a single set of registers. However many complex programs are difficult to express in a single threaded manner. This is particularly true for programs that involve asynchronous events, or when it is necessary to overlap computation and communication. In thread-based programs, there are multiple threads of control, and each thread may progress independent of other threads. Of course if there is only one processor, control needs to switch back and forth among these threads, under the control of some scheduler, and concurrency control mechanisms such as locks must be provided to allow threads to share data in a safe manner in spite of the interleaving of control among them. A threads package typically consists of three components: (1) a mechanism to *suspend* the execution of a running thread and *resume* the execution of a previously suspended thread; (2) a *scheduler* that manages the transfer of control among the threads; and (3) *concurrency control* mechanisms.

Many thread packages and standards have been developed in the past few years [10, 6]. However, the *gluing together* of scheduling, concurrency control and other features with the mechanisms to suspend and resume threads is problematic for the requirements of interoperability. E.g. the particular scheduling strategy provided by the threads package may not be appropriate for the problem at hand. Converse separates the capabilities of thread packages modularly. In particular, it provides the essential mechanisms for suspending and resuming threads as a separate component, which can be used with different thread schedulers and synchronization mechanisms, depending on the requirements of the parallel language or application [15].

Synchronization mechanisms:

Locks are implemented by having queues attached to each lock. If a lock can be obtained, the thread trying to obtain the lock continues (after setting the lock to its locked state). If not, the thread is suspended and placed in a queue for the lock. A thread which releases the lock causes the shifting of ownership of the lock to the first thread in this queue and awaken this thread so that it can continue executing when it is scheduled. *Condition variables* allow several threads to block on a single condition. Calls are provided for threads to wait on a condition variable, and for threads to either signal a condition variable, causing the unblocking of one of the threads, or to broadcast a condition variable, which causes the unblocking (i.e. awakening) of all the threads that are waiting on the condition variable.

3.5 Dynamic load balancing

The load on a processor is affected by modules in all languages, hence Converse supports load balancing across language modules. The need for load balancing arises in parallel programs in many contexts. A particular situation of interest is when the program creates a piece of work or a task that can be executed on any processor[1]. The load balancer assigns the task to a processor depending on the load measures on other processors at that point in the program.

A language runtime may hand over a "seed" for a task, in the form of a generalized message, to the load balancer on any processor. The load balancing module moves such seeds from processor to processor until it eventually hands over the seed to its handler on some destination processor. This module may interact with a local scheduler and may send messages to its counterparts on remote processors for exchanging load status information. It can also make calls to other entities for ascertaining the load on the local processors. Several load balancing modules are supported in Converse. Each one is often useful in a different situation. Depending on the application, the user is able to link in a different load balancing strategy.

[1] Other kinds of load balancing situations include dynamic object migration and quasi-dynamic load balancing. In object migration, entities such objects or individual threads are moved from one processor to another while the computation is in progress. In quasi-dynamic load balancing, after a phase or period of computation has completed, tasks are moved between processors to balance load. Both migration and quasi-dynamic load balancing can be implemented on top of Converse as Converse libraries. These, however, are beyond the scope of this paper.

4 Status and Performance

The basic Converse framework has been implemented on networks of Unix workstations connected by Ethernet/ATM, IBM SP, Intel Paragon, CM-5, Convex Exemplar, nCUBE/2, and on top of the Fast Messages layer [13] on the Cray T3D and Sun/Myrinet networks. Prototype implementations of PVM messaging and SM (a simple messaging layer) are complete and simple multi-lingual programs are demonstrated to run on the above machines. The Charm and Charm++ [11] parallel object-oriented languages have been retargeted for Converse. The machine interface of Converse is meant to be implemented at the lowest level on individual machines. On some machines the lowest and most efficient layers of the system were available to us. On other machines, it is necessary to secure the vendor's cooperation to implement the machine interface most efficiently.

Performance: The first set of performance experiments (Figure 4) involves simple message passing performance. This was measured using a round trip program that sends a large number of messages back and forth between two processors. Using this, the average time for one individual message send, transmission, receipt and handling was computed for various machines. On the receiving processor every message was delivered to a user-level handler which responded by sending a return message.

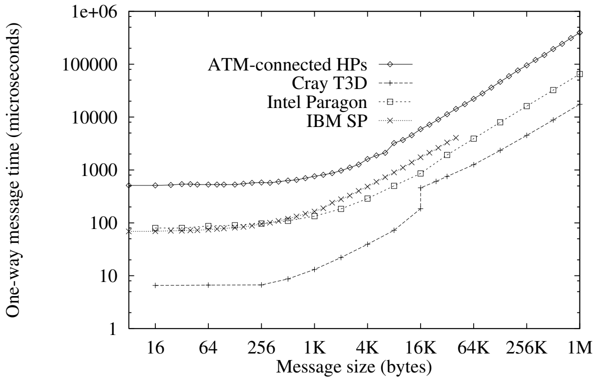


Figure 4: Message passing performance

Overall, the performance is almost as good as that of the lowest level communication layer available to us on these machines. For example, the FM library using Myrinet switches delivers messages up to 128 bytes in 25 microseconds, whereas Converse messages need about 31 microseconds. On the T3D, the performance is very close to the best possible on the Cray hardware for short messages. The jump at 16K bytes (Figure 4) for the Cray T3D is due to copying during packetization, which we believe can be eliminated.

In the second experiment, we incorporated queueing to investigate the overhead seen by languages using scheduling. Each handler upon receiving a message enqueues it in the scheduler's queue. The scheduler then picks a message from its queue and invokes its handler. This cost of scheduling is paid only by languages which use the queue for scheduling objects. This experiment was done only on one machine (Sun workstations connected by Myrinet switches — Figure 5) to illustrate the magnitude of scheduling overhead. The scheduling is seen to add about 9 to 15 microseconds for short messages. For large messages, the relative difference becomes negligible.

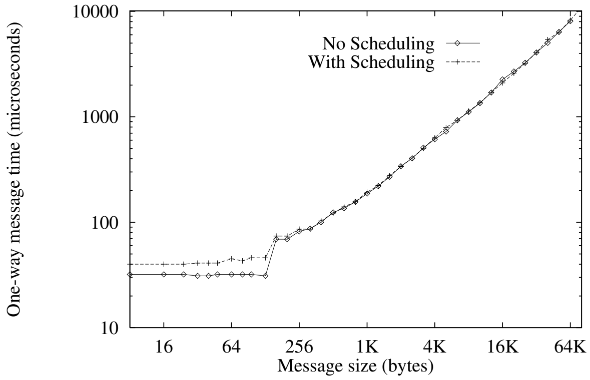


Figure 5: Scheduler Performance

Thus, although Converse provides a broad functionality, it achieves its objective of ensuring that languages and applications pay the overhead only for features that they use.

5 Summary and Future Work

We have presented the design and rationale for a comprehensive framework for supporting interoperability among a wide range of parallel languages and paradigms. The design is based on the fact that entities in different parallel languages can be classified into three basic categories from the point of view of the scheduling of the processor: (1) single-process modules which permit no concurrency and require programmers to transfer control among modules explicitly; (2) concurrent objects, and (3) threads, which both allow for concurrency and transfer control among their modules implicitly under the control of a scheduler. A unified scheduler which is exposed to language-specific runtime libraries and a generalized notion of messages allows these three basic paradigms to coexist. The thread object (which supports the thread abstraction without intertwining scheduling functionality), the generic message manager (which can be used to store and retrieve messages) and the load balancer (which balances load across language modules) further facilitate the design and implementation of individual language runtimes.

The Converse framework is useful in the following ways:

- Parallel programmers do not have to convert an entire application to a particular language. For each part of the application, the most suitable language or paradigm can be used.
- Pre-existing libraries written in different languages can be reused in a single application. For example, in a parallel molecular dynamics application [2] being developed using Charm++, we will be able to use an N-body module (based on the fast multiple algorithm), which happens to be written in PVM.
- Development of parallel languages, coordination languages, or libraries based on new paradigms is simplified because commonly used runtime modules such as a scheduler, a threads package, a message manager and a load balancer are provided. As an example, a small Chant-like [7] multi-threaded language that supports tagged messages was implemented with very little effort using Converse, because of the functionality provided by the above components.

Although we are convinced of the breadth and flexibility of the Converse design, it is clear that additional research and implementation effort is needed for Converse to fulfill its promise — that of supporting interoperability between a wide variety of languages without loss of efficiency. Some of the features we are currently working on include shared memory operations (*put* and *get*), pre-emptive messages, parallel file I/O, object-orientation, scatter and gather based communication, and group communication.

Acknowledgements: The authors would like to thank Terry Allen, Robert Brunner and Attila Gursoy for their help in preparing this paper. We would also like to thank the Pittsburgh Supercomputing Center, Argonne National Laboratory, and Prof. Andrew Chien and Prof. Klaus Schulten at the University of Illinois for the use of their computing facilities.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] J. Board, L. V. Kale, K. Schulten, R. Skeel, and T. Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[3] F. Bodin, P. Beckman, D. Gannon, and S. Narayana, S. an d Yang. Distributed pC++: Basic Ideas for an Object Parallel Langua ge. *Scientific Programming*, 2(3), 1993.

[4] K.M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *Proceedings of the Fourth Workshop on Parallel Computing and Compilers.* Springer-Verlag, 1992.

[5] A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[6] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[7] M. Hainer, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing '94*, November 1994.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, January 1993.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiler support for machine independent parallel programming in Fortran-D.* Elsevier Science Publishers B.V., 1992.

[10] Draft Standard for Information Technology—Portable Operat ing Systems Interface (Posix), September 1994.

[11] L.V. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[12] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, November 1993.

[13] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Proceedings of Supercomputing 1995*, dec 1995.

[14] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[15] J. Yelon and L. V. Kalé. Thread primitives for an interoperable multiprocessor environment. Technical Report 95-15, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1995. Submitted for publication.