MICE: A Prototype MPI Implementation in Converse Environment

Milind A. Bhandarkar and Laxmikant V. Kalé
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{milind,kale}@cs.uiuc.edu

Abstract

This paper describes MICE, a prototype implementation of MPI on the Converse interoperable parallel programming environment. It is based on MPICH, a public-domain implementation of MPI and uses the Abstract Device Interface (ADI) which has been retargeted on top of Converse. MICE makes use of message-managers and allows use of thread-objects to let MPI modules co-exist with other types of computations and communication (such as a library computation in Charm++ or asynchronous computations in multipol) within a single application. It also makes it possible to interoperate PVM (in a restricted form) and MPI modules. Thread-objects make it possible to build multi-threaded MPI programs. This MPI implementation demonstrates that it is possible to provide interoperability without any significant performance degradation.

1. Introduction

Message-Passing parallel programming has become the method of choice for many parallel programmers. There have been tremendous advances in this field in the past few years. One such advance is the emergence of Message Passing Interface Standard, MPI[5]. Since the detailed specification of MPI 1.0, many public-domain and vendor-specific implementations of MPI became available and many existing applications were ported using MPI. Many new applications are also being developed in the MPI environment. However, there aren't many tools available for converting existing applications from other parallel programming environments to MPI. This may be straightforward for some programming environments but could be very difficult for others. Also, there is currently no way to use modules developed using other languages (or run-time systems) in MPI programs. In particular, we are interested in ways to use modules developed using message-driven objects alongwith MPI modules in a single application.

This paper describes MICE, a prototype implementation of MPI on the Converse[3] interoperable parallel programming environment. It is based on MPICH[1] implementation and uses the Abstract Device Interface (ADI)[7] which has been implemented on top of Converse. Converse makes it feasible to employ modules written using different parallel programming languages and run-time systems in a single application. Currently, in addition to MPI, a number of parallel programming environments such as threaded-PVM[6] (without process management facilities), Charm[2] (an object-based message-driven language), Charm++[4] (an object-oriented message-driven language), IMPORT (a parallel simulation language) and threaded SM (a simple messaging layer) have been ported to Converse. Other parallel programming languages and libraries such as POL (Parallel Object Language), multipol library, and CRL (an explicit distributed shared memory environment) are also being ported to Converse.

The next section describes Converse. MPICH device interface is described in section 3. In section 4, we present our implementation of MICE. Section 5 describes how MPI could be used with modules written in other languages. We compare the performance of our MPI implementation with MPICH in section 6 and conclude in section 7.

2. Converse

Converse[3] is an interoperable parallel programming framework for implementing parallel languages. Converse has a dual objective to first, support co-existence of modules written in different parallel languages/paradigms in a single application program; and second, to support quick development of runtime systems for new languages and parallel libraries. It was designed to support the three important classes of parallel languages: SPMD, Message-driven objects and multi-threaded languages. Data parallel languages could be implemented in any of these classes and therefore are not considered as a separate category.

Converse has a component-based architecture that has a desirable feature of need-based cost. (Languages pay only for the features they use.) Also, it allows user to override any of its components. Some of the major components of Converse are its scheduler, machine interface (CMI), thread-objects, message-manager and load-balancing strategies.

Converse employs a notion of generalized messages to utilize a single scheduler component for all the entities in an application. A generalized message is a contiguous block of data that contains the handler index in the first few bytes. This handler is invoked when the message is dequeued from the network and is processed. The handler can then grab the ownership of this message from the run-time system and process the message immediately or may decide to enqueue it for future processing.

Converse scheduler repeatedly picks up the generalized messages from its queue and calls their handlers. When confronted with multiple choices in available messages, the scheduler has to decide which one to choose for processing. This decision may have a major impact on the performance of applications such as branch-and-bound search. Converse allows the language runtime to associate priorities with generalized messages and provides different prioritized queuing strategies.

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 priorities.

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. MPI also provides such a portable interface. However, it represents 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 the delivery sequence is unnecessary for many applications. Also, as we have shown in section 5, MPI receive could be easily implemented on top of CMI and other language modules do not have to pay the overhead for maintaining order or for a complicated message retrieval.

In addition to the minimal machine interface, Converse specifies an extended machine interface (EMI) that allows efficient implementations for operations such as vector send and *get* and *put* operations on global pointers.

In the implementation of MICE, we have made extensive use of message-managers, which are provided as a library in Converse. 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. 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 any tag field. The message manager can be used by multi-threaded as well as single-threaded modules.

Each process belonging to a parallel application could be single threaded or multi-threaded. Most thread-packages combine the ability to create, suspend and resume a thread, scheduling of threads as well as synchronization mechanisms into a single monolithic component. However, language implementors need a finer control over each of these components. Therefore, Converse modularly separates these three capabilities and exposes them to the language implementors. Using Converse, one can create thread-objects[8], schedule them under one's control without depending on the system's scheduler. (If one wishes to use the scheduling provided by the system, one is free to use it though.) Various synchronization mechanisms such as locks and condition variables are supplied by Converse, as a component, which could be overridden by the user.

Converse uses a single scheduler for scheduling computations specified by messages as well as threads. That is, generalized messages are enqueued in the scheduler's queue for awakening suspended threads. Scheduler treats these messages the same way as it treats other messages. This makes it possible to have elaborate prioritization schemes even for threads.

Converse supports load balancing across modules. Its design is particularly influenced by situations where a piece of work created on one processor may be executed on any processor, preferably with the least load. To achieve this, the language run-time hands over a "seed" for new work created in the form of a generalized message to the load balancing module which then moves the seed around the network until any processor decides to enqueue it to its local queue and process it. Converse has many built-in load balancing strategies which could be selected by the user at link-time. (Or the language implementor could provide the user with more choices of load balancing strategies.)

3. MPICH Device Interface

MPICH[1] is a public-domain implementation developed at Argonne National Laboratory and Mississippi State University. This is a two level implementation of which the top layer provides device-independent functionality of MPI and the bottom layer implements the device-specific part. There are "hooks" from the device-independent part into the device-specific part. The device-specific part is called the Abstract Device Interface (ADI)[7]. The ADI itself is a multi-level interface. It consists of certain core routines that must be implemented, and a set of extension routines that are implemented if the underlying machine supports them and has better performance than the corresponding MPICH routines. However, these extension routines do not add any more functionality and the implementation that supplies only the core ADI functions is a complete implementation of MPI.

The core of the ADI consists of routines to post and complete communication requests, testing for availability of messages, initialization and termination of ADI, getting rank of calling process, and size of the participating processes. The extension routines include routines for collective operations as well as blocking sends and receives. Most ADI routines are passed the ADI *context* returned by the `MPID_Init` subroutine. Based on this context, the ADI can select routines to call. In our current implementation, we do not use the ADI context, but plan on using one in the future to allow heterogeneity. This ADI interface is constantly evolving, but we do not foresee any incompatible changes that we may not be able to implement using Converse.

4. Implementation

MICE is implemented by developing the Abstract Device Interface (ADI) of MPICH using Converse primitives. As described in the previous section, the ADI is a set of device-specific routines that the device-independent part of MPICH uses to post and complete communication requests as well as to enquire about participating processes.

Our implementation of the ADI uses a separate Converse handler for MPI messages. This handler is registered in `MPID_Init`. Whenever the machine layer of Converse (CMI) receives message (either from the network or from the local processor), it invokes this handler. This handler then sees if the message was expected according to the sequence number associated with the message. (Converse does not guarantee in-order delivery of messages. Therefore, to ensure order among a pair of processors, we need to associate a sequence number with a message. A message with the "next" sequence number per source processor is always "expected" by the ADI.) If the message arrives out of order, it is buffered until all the previously dispatched messages arrive, at which point it is moved into the message manager. The ADI routine for testing for message arrival simply probes the message manager (using the `CmmProbe` function provided in the message manager component of Converse.)

For explicit control-regime interoperability (discussed in the next section), or in standalone mode, MPI implementation is single-threaded. We wait for messages directed at the ADI message handler whenever we block for any message in the `MPID_Blocking_recv` call by calling `CmiDeliverSpecificMsg` with the ADI handler index as a parameter. This function blocks until a message intended for the specified handler arrives from the network and calls the ADI handler function. For implicit control-regime interoperability, MPI module is run as a thread object. Whenever MPI needs to wait for a message to arrive, we suspend the MPI thread-object and transfer control to the Converse scheduler, which wakes up the MPI thread whenever an MPI message arrives. The choice between these two implementations of MPI library can be made at the link time. For implicit control-regime interoperability, we had to write a variant of Converse scheduler that exits when there are no more messages to be processed. The component-based architecture of Converse permits us to rewrite this scheduler without rewriting any other component.

Since most existing MPI programs depend on ability of doing standard input and output from any node on the system (though MPI does not demand it), we have overridden the system routines `printf`, `fprintf`, `scanf`, and `fscanf` with our own routines coded as a part of the MICE library, which make calls to the Converse routines `CmiPrintf` and `CmiScanf`. One important change that the users of MICE will have to do to link their MPI programs is to change the name of the `main` routine to `user_main`. This should be done, because Converse supplies the `main` routine that could do some internal data structure initializations before calling `user_main`.

MICE became operational recently and has passed the entire test suite (about 150 MPI programs) supplied with the MPICH distribution on network of Solaris 2.4 based workstations. Since Converse is portable across many architectures, we expect MICE to run on those machines as well. However, we have not performed elaborate testing on any other machines. (We foresee some minor modifications to be made in the top layer of MPICH implementation while porting MICE to the shared memory machines supported by Converse.) Also, MICE currently works only with C programs. We plan to provide MICE for FORTRAN in future.

We carried out several performance tests (included in the MPICH distribution) and the results are reported in section 6. Most of the ADI routines that directly map onto the Converse routines are are provided as macros to avoid extra function call overhead. We also use Converse's vector send capabilities to attach header to a message without having to copy the header and message into a contiguous system buffer before calling `CmiAsyncSend`. There are some more performance optimizations that we plan to do in near future.

5. Interoperability

Two important issues considered while designing Converse for supporting interoperability were: Degree of concurrency allowed in a language and the control regime. While MPI does not restrict the degree of concurrency allowed in an application, neither does it assume that concurrent computations are active. Therefore, it is the programmer's responsibility to manage concurrency in an MPI application. Some traditional SPMD environments do not allow multiple threads of control inside an application. At the other end of the spectrum are the message-driven languages such as Charm and Charm++, where the run-time system is free to choose which methods (and in which objects) to invoke depending on the availability of messages directed at them. An application could have multiple threads of control and the run-time system for such languages usually provides ways to let the application influence the order in which these threads will be scheduled for execution. The second issue is that of a control-regime, that is, how control is transferred between individual modules. If multiple modules within a single application can execute in an interlaced manner on multiple processors, it is referred to as an implicit control-regime. If at any time only one module is active across all processors, and control is transferred from one module to another explicitly by the application, it is called an explicit control-regime. Figure 1 from [3] illustrates control transfers in explicit and implicit control-regimes in a multilingual program.

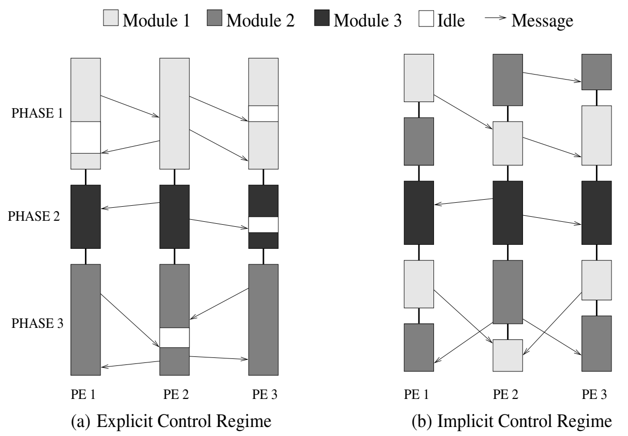

Figure 1. Control-Regimes for Parallel Programs

MICE supports both explicit and implicit control-regimes and also allows MPI modules to co-exist with other languages with different levels of concurrency.

Let us consider a hypothetical example of an application which has two functionally separate modules. One module solves a system of linear equations and the other performs adaptive quadrature scheme of numerical integration. Suppose that a linear system solver module is already available in MPI. Adaptive quadrature schemes require dynamic load balancing and a program that performs numerical integration using this scheme has already been written using a language such as Charm that supports dynamic load balancing. (In adaptive quadrature scheme, the interval is divided until the number of samples needed to be taken on that interval are sufficiently small. If in some interval, the variance of the function to be integrated is small, it is not divided any further. Thus the number of intervals created are not known until run-time. Therefore dynamic load balancing is needed to implement this scheme efficiently.) If one plans to build a system combining these two modules where the linear system is formed based on the results of the numerical integration and is solved using the linear system solver; clearly, one does not need both the modules to execute simultaneously. This can be programmed using explicit control-regime interoperability. We start this application by initializing the Converse runtime and sending a message to the starting entry point of the Charm module. We then start the scheduler with (-1) as an argument so that the control will return to the application after the Charm module has terminated. (Converse provides a call to terminate the scheduler, which could be used to return control back to the caller.) After the control is back with the application, it calls the MPI module to form and solve the linear system. Figure 2 gives example of such code. The function `CallCharmModule()` in this code passes parameters in the form of an initialization message to the initial entry-point of the Charm module. It then starts a scheduler, which processes the initialization message by invoking the initial method of the main object in the Charm module. After performing the numerical integration, the Charm module stores the result in some location and makes a Converse call to exit the scheduler. At this point the control returns to `CallCharmModule` which then picks up the result and returns it to the application. The result of the computation performed by the first module has to be shared with the MPI module using shared space on some processor(s). (Not coincidentally, the processor indices in `MPI_COMM_WORLD` are the same as processor indices in Charm.) We plan to provide more portable and elaborate mechanisms for inter-module data transfer in future.

As another hypothetical example, let us consider a situation where both the Charm and MPI modules described above are to be executed concurrently. (Let's say we need to compute numerical integration using Charm module and we want to compute the approximate correction factor by solving a linear system.) In this case, implicit control-regime interoperabilty is the method of choice. This application initializes MICE (which in turn initializes Converse) and then sends a message to trigger the computation in Charm module. (Currently, this should be done by providing a special function written in the Charm module, because the message accepted by an entry point in Charm is Charm's internal object.) Instead of processing the message just sent by calling the scheduler, our application now continues with the MPI code. However, all the blocking communication calls in MICE contain calls to the scheduler to process available messages according to their priorities and return when no messages are available. Thus when MPI module is blocked on certain message, the run-time system schedules other work in Charm module, allowing concurrent computation in both modules. Since Converse uses messages with handlers specified inside them, and since handlers are private to a module, the communication spaces of MICE and Charm are not allowed to overlap. Thus the condition of progress is satisfied in MICE. Fairness condition is also satisfied within a module. More work needs to be done to provide fairness across modules by making MICE messages include priority fields and scheduling them through the scheduler according to their priorities.

6. Performance

We conducted several performance tests on our implementation of MPI. These test programs are part of the MPICH distribution. In particular, we were interested in the communication overhead introduced by Converse as compared to an optimized version based on the P4 runtime system. We ran these tests on network of solaris workstations. In order to eliminate the large variance in results due to varying network load, we created all the processes of an application on the same standalone workstation. The test programs we ran measure the round trip time for a message between two processes for different sizes of messages. We ran the tests for blocking as well as non-blocking send and receive routines. Also, we tested both implementations for short as well as long messages. For measuring the latency and bandwidth for short messages, the message size is varied from 0 to 1024. For long messages, the message size is varied between 16KB and 64 KB. The rsults are shown in Table 3, and figures 4, 5, 6, and 7. The round-trip performance of MICE is comparable to that of the ch_p4 version for short messages. For long messages the performance of MICE is slightly worse than the ch_p4 version. However, we need to mention here that the current version of MICE is a preliminary implementation and that there are numerous optimizations that will be done in near future to make it more efficient.

7. Conclusion and Future Work

We have implemented MPI using the primitives provided by the Converse interoperable runtime system. The performance of our implementation is comparable, although slightly lower, than MPICH implemented on a workstation-network architecture, as expected from a preliminary implementation. At this performance level, clearly, the converse implementation of MPI can be used in applications without degrading their performance significantly. More important, we have demonstrated multilingual interoperability: MPI modules can now be used in conjunction with modules written in Charm++ or threaded-message-passing libraries. In addition, our implementation achieves portability due to the portability of the Converse Machine Interface. Our implementation is thread-safe when used in conjunction with thread-objects provided by Converse.

More detailed performance studies need to be done in the presence of multilingual modules. Utility of such interoperable environments needs to be investigated further. Mechanisms for transfer of data across modules written in different languages need to be designed and implemented.

References

[1] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[2] L. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Proces sing*, Aug. 1990.

[3] L. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *International Parallel Processing Symposium 1996 (to appear)*, 1996.

[4] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, 1993.

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

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

[7] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*, August 1995.

[8] 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, Dec. 1995. Submitted for publication.

```
int user_main(argc, argv)
{
 MPI_Init(argc, argv);
 // Initialize Charm
 CharmInit();
 // some application-specific initialization
 Initialize();
 // Deposit start-message for Charm module
 // and start the scheduler
 result = CallCharmModule();
 // Resume MPI Module
 FormLinearSystem(result);
 SolveLinearSystem();
 PrintResults();
 MPI_Finalize();
}
```

Figure 2. Pseudo-code for an interoperable program

	ch_p4		Converse	
	Latency (μSec)	Bandwidth (MB/s)	Latency (μSec)	Bandwidth (MB/s)
sblck	573.0	9.93	687.37	10.14
snblck	648.9	10.68	657.10	10.04
lblck	867.0	13.91	1034.56	9.29
lnblck	785.32	12.96	809.61	9.06

Figure 3. Performance comparison of ch_p4 and MICE. sblck = blocking, short messages; lblck = blocking, long messages; snblck = nonblocking, short messages; lnblck = non-blocking, long messages.

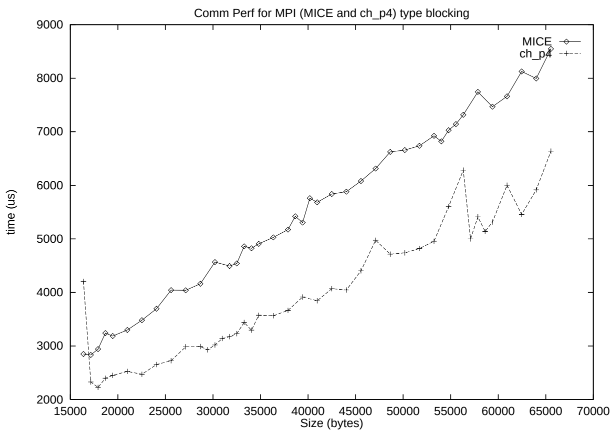

Figure 4. Point-to-point long blocking Messages

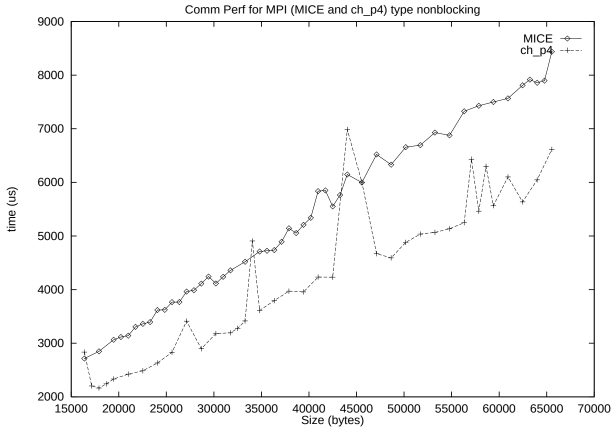

Figure 5. Point-to-point long non-blocking messages

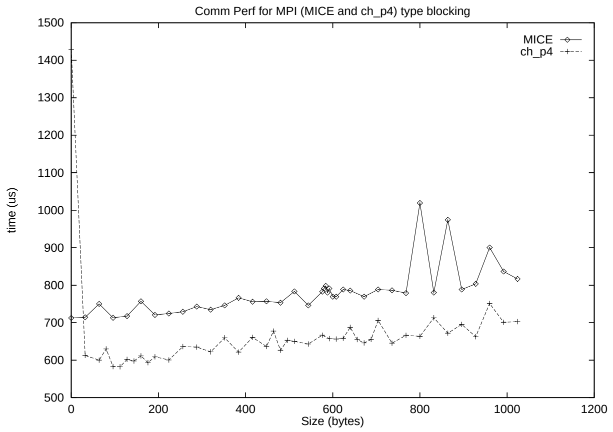

Figure 6. Point-to-point short blocking messages

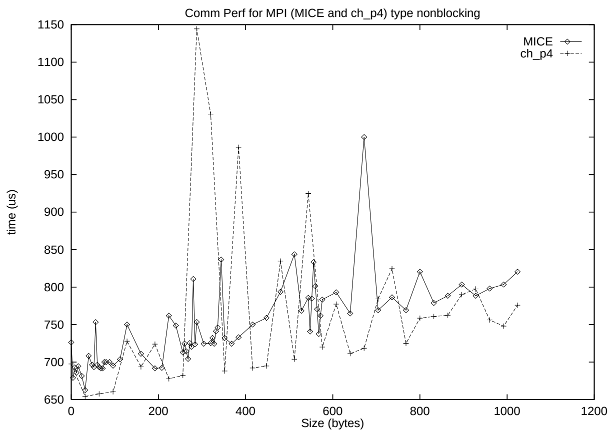

Figure 7. Point-to-point short nonblocking messages