Implementation of Converse Interoperable Parallel Programming Environment on Parsytec CC Multicomputers*

Murat Atun, İlker Cengiz, Reşat Şireli and Attila Gürsoy
Computer Engineering and Information Sciences
Bilkent University Ankara, Turkey
{atun,icengiz,sireli,agursoy}@cs.bilkent.edu.tr

Abstract. Asynchronous message handling is an approach to reduce communication delays in message passing parallel architectures. Converse is an interoperable parallel programming environment which combines asynchronous message handling and user level scheduler. It supports message-driven style of programming where messages trigger computations as opposed to explicit receive of messages in traditional message-passing programming. In this paper, we discuss an efficient implementation of Converse environment on Parsytec CC distributed memory machines and present performance results of a few collective communication operations implemented in message-driven (using Converse) and in message passing styles.

1 Introduction

The delays due to communication are the major source of performance degradation in message passing parallel computers. These delays are not only due to latencies in the communication network, a significant portion of it is due to the software overhead to handle sending and receiving messages from user level to the hardware level. Asynchronous message handling is one of the approaches to reduce this delay [6]. In contrast to message passing, in asynchronous message handling model, message reception results in invocation of a function, the message handler. There are several asynchronous message-handling systems developed [2, 7]. They vary in the way the handlers are executed, the expressiveness of the model etc. Active messages is one of the implementations of asynchronous message handling. An active message consists of a handler function address and data. On receipt of an active message, the processor extracts the handler address and invokes the handler with the data as parameter. Active messages put certain restrictions on handler execution such as it does not allow blocking the handler. Other mechanisms include single-threaded up-call and popup-threads, which disallow all blocking by letting message handlers use locks to synchronise their shared-data accesses. Asynchronous message handling deals with only reducing the software overhead of sending and receiving a message. However, the physical reality dictates that accessing remote information will always be slower than accessing the local one. Hence, in addition to reducing the delays, overlapping these delays due to remote data access with useful computation is also important for improving performance of parallel systems. Message-driven execution is a promising model in this sense [8]. In message-driven execution, there are many objects per processor. When a message arrives for an

*This work was partially supported by TÜBİTAK under the grant EEE-AG 247.

object, the object is eventually activated with the message. Although message-driven execution sounds similar to asynchronous message handling, it requires a scheduler. In asynchronous message handling the invocation of handlers is not under the control of user and they lack user level scheduler. Converse [5] is one of the earlier parallel programming systems that combines scheduler with asynchronous message handling.

In this paper, we will discuss implementation of Converse to Parsytec CC parallel computer and performance of some collective communication operations implemented with machine's native message passing libraries and with the Converse environment. The motivation behind this work is to port Converse system as efficiently as possible to Parsytec computer in order to use the advantages of message-driven execution in applications that will be developed later. The paper is organized as follows: Section 2 gives brief information on Parsytec CC systems followed by Converse and its machine interface in Section 3. Section 4 summarizes the design decisions and implementation of Converse-Parsytec interface layer. We present the performance results of ring, k-to-all and all-to-all communication operations for message-passing libraries available on the machine (EPX, PVM) and for Converse environment, and conclude in Section 6.

2 Parsytec CC System

The Parsytec CC system is a distributed memory MIMD computer manufactured by Parsytec GbmH in Aachen, Germany. The processors are connected via a high speed communication switch which uses wormhole packet routing mechanism. The system used in this work has 24 nodes with 133 MHz PowerPC 604 microprocessors. The communication switch has a 27 MB/s sustained speed for point to point communication. All nodes of Parsytec CC system run the AIX operating system with EPX (Embedded Parix, the native message passing library). EPX supports both synchronous and asynchronous message passing over virtual links between processor nodes. In synchronous communication, (`Send()`, `Recv()`), sender process is blocked until the receiver process issues a receive for the message. Asynchronous communication is accomplished by creating a thread to perform the communication concurrently without blocking the sender thread. The asynchronous message-passing functions include `ASend()`, `ARecv`, and `ASync()`. `ASync()` function is used to wait for the completion of communication on a link. More information on EPX can be found in EPX [3].

3 Converse and its Machine Interface (CMI)

Converse [5] is an interoperable parallel programming framework for implementing parallel languages and low level parallel computation kernel/libraries. It is designed to support quick development of new parallel programming paradigms and interoperability between multi-paradigm components within a single application. The machine model consists of a set of processors that communicate via messages. A message contains a function pointer (handler) and user data. Each processor runs a scheduler which picks the next message from the message queue and invokes the appropriate handler for the message. Note that there is no message receive primitive in Converse programming interface. Several parallel programming languages and libraries have been implemented with Converse including MPI [13] and PVM message-passing libraries, Charm and Charm++[9] message-driven object-based parallel programming languages, and a few more experimental languages. Further details on Converse programming framework and parallel languages can be found in `http://charm.cs.uiuc.edu` and in [1]. Converse has a component-based architecture. One of its major components is the machine interface (CMI) which is the layer between Converse runtime and the parallel machine. Porting Converse to a new machine requires only the implementation of the CMI module on the new machine.

3.1 Converse Machine Interface — CMI

CMI defines a minimal interface between the machine independent part of the Converse runtime such as scheduler and the machine dependent part which differs from system to system. CMI is responsible for process creation, process coordination at the initiation and termination points, and communication. The major groups of CMI functions are:

- Message sending functions: These are synchronous and asynchronous message sending operations. `CmiSyncSend()` returns as soon as the message is entered into the network. `CmiAsyncSend()` is the asynchronous one which returns almost instantaneously. The completion of an asynchronous message send operation is checked with the `CmyAsyncMsgSent()`.

- Initialization/termination functions: These functions create processes on every processor, initialize communication between processes, and also provide graceful termination and cleanup operations.

Other machine-specific utilities in the CMI layer are synchronization primitives, dynamic memory allocation, timer, and input/output function.

4 Implementation

The major part of the CMI functions are the communication related ones. These functions can be easily implemented if the underlying machine provides a communication interface that fits to CMI. The easily accessible low level communication interface on Parsytec machine is the EPX message passing library that provides a set of communication and thread operations. During the implementation work, however, we have found out that the communication primitives of EPX were not sufficient to express directly the functionality required by the CMI. In this section, we will describe an efficient layer between CMI and EPX which handles the mismatch between these two layers. The major mismatch points are listed below:

- Synchronous send: CMI synchronous send functions return the control to the caller when the message is entered into the network but not necessarily received at the destination. EPX synchronous send (`Send()`) function, on the other hand, blocks the caller until the receive operation is invoked at the destination processor. Therefore, EPX synchronous calls cannot be used to implement synchronous primitives of CMI. Although, this problem can be solved using some other forms of send operation provided by EPX (`SendLink()`), we preferred to provide a complete solution that handles the other problems that we encountered.

- Asynchronous send: CMI asynchronous send functions return the control to the caller immediately with a `request-id`. The completion of the send operation can be checked later by using the `request-id`, and the message cannot be reused until completion. However, EPX library does not have primitives for querying individual asynchronous send operations for completion. The only facility available is to check the completion of all messages on a particular link.

- Asynchronous receive: In Converse, messages coming from remote processors are put into a queue where the scheduler processes them. Messages can arrive in any order. EPX receive primitives however need to know in advance the source and the size (or maximum size) of the message to be received. In order to be able to receive messages without knowing their sizes, a message-probe operation with the ability to ask the size of the message to be received next is necessary.

- Potential performance bottleneck with asynchronous communication: EPX creates a thread for each asynchronous message send or receive operation. This might cause significant performance penalty if the application involves many messages. Although EPX allows us to limit the number of threads, one cannot control which thread is dealing with which message. A thread might be working on two messages with different destinations. This might limit possible overlap of communication.

The first part of the implementation deals with initialization. In startup phase, one process on each processor is created. Then, a communication topology is established, which connects any two pair of processors using `MakeLink()` utility of EPX. The second part implements CMI communication operations. To overcome the problems related with asynchronous message sending and receiving, we have designed a layer between CMI and EPX which utilizes a fixed number of dedicated communication threads:

- Sender threads: For each destination processor, there exist one sender thread associated with a send-message queue. The send-message queue is used to store asynchronous messages until their transmission are successfully started. Whenever the sender thread is idle and there are messages in its queue, the thread picks the next message and sends it synchronously (`SendLink()`). Once the message is entered into network, it is marked as "sent". `CmiAsyncMsgSent()` simply checks this flag for completion of the transfer. If the send-message queue is empty, the sender thread suspends itself not to consume CPU time until the main thread (scheduler) puts another message into the send-queue.

- Receiver threads: The incoming messages are stored in the schedulers message queue. A receiver thread checks the incoming transmission links, whom it's dedicated for, and as it detects a message on any of those links, it receives the message using `RecvLink()` primitive then it places the incoming message into the receive-queue of its owner processor. When a message is detected in receive-queue, it's picked up rapidly by the scheduler using `CmiGetNonLocal()` function and associated handler function is invoked with the message itself as a parameter. For message size problem, we developed an efficient mechanism to receive arbitrary size messages by defining a standard message size SIZE. The mechanism works as follows; if the length of a message does not exceed SIZE value, it is sent as a single message. Otherwise, it sliced into two pieces, basically the total size of the message is included in the first piece, then the rest is sent as a whole. In such a case the receiver detects that the incoming message is not complete, and receives the rest of the message to be combined with the first slice received.

In our implementation, we have chosen N-sender/1-receiver case (that is, one sender thread for each possible destination processor, and one receiver thread), where the other alternatives included 1/1, N/1, 1/N, and N/N sender/receiver threads. We found it better to use N send-queues and N sender treads where each thread is associated with a separate message-queue and a processor, in order to avoid contention on queues on concurrent send requests to different target processors. For the receiving threads, however, we needed to maintain a single queue for incoming messages to keep the First-In- First-Out (FIFO) order. Using one receive thread results better than using N receive threads, since in the latter case the threads will waste CPU time sharing the single receive-queue. The one-and-only receive thread uses the `SelectList()` function of EPX (a function to select incoming messages non-deterministically) in order listen to all incoming transmission links.

5 Performance Results

We conducted a performance study in order to measure and compare the performance of our Converse implementation as well as asynchronous message handling and message-driven programming (that Converse framework supports naturally) and the message passing programming (supported by EPX and PVM libraries[4]). Converse is designed and implemented to be an underlying system for other parallel programming systems and parallel computation kernels. This fact results Converse to have an inappropriate nature for implementing complex applications directly. Hence our benchmark computations were chosen to be basic collective communication operations that can be viewed as a parallel communication library to be used by the applications. A main driver program (application) repeatedly invoked the benchmark communication operations. The study included a simple ring communication algorithm for measuring message latency as observed by the programmer which is also the general choice for bandwidth latency problems, k-to-all broadcast and all-to-all personalised communication algorithms.

The algorithms have been implemented using EPX, PVM, and Converse systems. All cases have been implemented as efficient as possible for 3 different message sizes of 64, 64K and 512K. Since Converse is built on top of EPX, what we achieve using Converse is possible with EPX (i.e., emulation of message-driven programming). But in order to compare the message-passing and message-driven approaches, we used the native programming styles of EPX and PVM (message-passing), and Converse (message-driven). The message-driven implementation allowed the program to pipeline consecutive operations dynamically. Such performance advantages of message-driven execution are discussed with detail in [12] particularly in applications based on multiple modules (library) invocations.

5.1 Ring

In the ring program, the processors are connected in a ring topology, and they pass messages from their predecessors to their successors. When the processor 0, that initiates the execution receives a message from its predecessor computation finishes for one step. The communication is regular in the sense that there is only one message and each processor knows from where to receive and to where to send the message. Table 1 shows the time required for messages of different length for the three versions of the ring. The EPX version is slightly faster than Converse and PVM versions for small message size. Since the latter systems are built on EPX, the cost introduced by Converse and PVM exceeds the time needed for small messages. However, as the message size gets larger, Converse results starts becoming better and gets closer to those of EPX. So, Converse incurs negligible overhead with its asynchronous message handling capability.

Message size	# of processors	Converse	PVM	EPX
64 Bytes	2	173	168	81
	4	355	339	118
	8	721	661	241
	16	1466	1339	488
64 KBytes	2	984	1113	713
	4	1964	2078	1369
	8	3919	4064	2753
	16	7902	8122	5466
512 KBytes	2	5490	6950	5230
	4	10930	13630	10290
	8	21770	26240	20690
	16	43710	53070	40960

Table 1: Timings for ring program (in msec)

5.2 K-to-all Broadcast (KABC)

K-to-all broadcast is an operation in which k processors simultaneously perform a one-to-all broadcast (OABC). Some parallel algorithms require a number of processors to send identical data to all other processors. Basically special cases of KABC occur as one-to-all broadcast when $k = 1$, and all-to-all broadcast (AABC)when $k = p$. OABC and AABC are used in several important parallel algorithms including matrix operations, Gaussian elimination, and shortest paths. A naive way to perform KABC is to sequentially send $p-1$ messages from the source to the other $p-1$ processors. In order to avoid redundant transmissions, we used a spanning tree approach. The source of the broadcast is the root of the tree. In first step the root sends the message to its left and right child on the tree. Upon receiving the message every processor makes a copy of the message, and then forwards it to its descendants on the tree. This example shows a significant improvement in case of handling asynchronous messages. In our application each source processor sends the same message to every other processor, but different processors broadcast different messages. Such a KABC may be followed by computations depending on the received messages, and new KABCs are initiated. At each KABC operation, k may change as well as the processors that are involved in the operation as initiators. In EPX & PVM versions, messages are handled in a fixed order. This is due the nature of EPX & PVM, they provide a message-passing environment (without emulating message-driven style). In EPX & PVM versions `RecvLink()` call is used for each message expected. Hence, even if the order of received messages is not important, the set of receive calls will somehow be executed in a fixed order. However in Converse version the messages triggers the handler functions, as they arrive. Table 2-(a), Table 2-(b), and Table 3-(a) display the results gathered from the three versions of KABC with varying message sizes and k values. Source processor wants to send the same message to every other processor, but different processors may broadcast different messages. The result-tables demonstrate, what Converse provides with its asynchronous message handling capability. As it was in the ring application, for small message size Converse performs at most equally with EPX, while PVM follows both of them. The increase in message size exposes the advantage of asynchronous message handling. These results ensure us about the necessity of the capability of asynchronous message handling. For KABC in 8 processors the difference between Converse and EPX versions comes out with no doubt.

(a) Message Size 64Bytes				
# PEs	K	Converse	PVM	EPX
2	2	172	292	83
4	2	766	378	168
	3	456	486	242
	4	519	599	323
8	2	623	561	300
	4	798	900	691
	6	964	1347	978
	8	1142	1689	1343
16	4	956	1369	699
	8	1746	2270	1339
	12	2109	3123	2008
	16	2468	3756	2615

(b) Message Size 64KBytes				
# PEs	K	Converse	PVM	EPX
2	2	861	1237	841
4	2	2004	2512	1638
	3	2593	3552	2501
	4	3298	4719	3663
8	2	3455	4813	3826
	4	6377	9588	9135
	6	8239	13928	13247
	8	10459	18877	17949
16	4	7957	13198	8727
	8	15598	23200	17456
	12	20898	34041	25991
	16	26876	44325	40203

Table 2: Timings for k-to-all broadcast for message sizes 64Bytes and 64KBytes (in msec)

(a) k-to-all broadcast (Message Size 512KBytes)				
# PE	K	Converse	PVM	EPX
2	2	4984	7290	5901
4	2	12178	16244	12873
	3	16583	23105	19826
	4	17197	32144	26389
8	2	22837	30593	27324
	4	41271	60796	64530
	6	55338	90821	92444
	8	70217	117521	127735
16	4	52246	82367	64216
	8	101657	141610	130525
	12	141633	218264	191889
	16	185829	280106	250256

(b) all-to-all personalized communication (message size in bytes)				
Msg size	# PEs	Converse	PVM	EPX
64	2	327	370	121
	4	754	764	583
	8	1503	3614	2423
	16	3044	11416	10051
64K	2	2648	3870	3303
	4	13570	17081	16674
	8	42164	64303	67659
512K	2	18862	36971	25828
	4	97160	122738	131118
	8	332548	648312	533513

Table 3: Timings for k-to-all broadcast and all-to-all personalized communication (in msec)

5.3 All-to-all Personalized Communication (AAPC)

In all-to-all personalized communication, also known as total exchange, each processor sends a distinct message to every other processor. In the literature this operation is used in parallel fast Fourier transform, matrix transpose and some parallel database join operations. Two versions of this program were implemented. In the first version it is assumed that AAPC is performed on a hypercube topology and messages allow an associative operation and are sent using store-and-forward routing. Communication takes place in $\log p$ steps where p is the number of processors. Pairs of processors exchange data in a different dimension in each step. Recall that in an p processor hypercube, a set of $p/2$ links in the same dimension connects two sub-cubes of $p/2$ processors each. At first step every processor holds messages of size $n/2$ and in successive steps message sizes decrease by its half, since after receiving, every processor performs a folding operation on its old message and the one that it received. In the other version, it is assumed that an associative operation is not allowed, and so each processor sends its special message directly to its recipients and later waits for its messages. Upon receive of any message, since the originator is known, each processor places this message to an associated local storage area. Results for first version are similar to ring results (PVM and EPX performs well since its a too synchronised algorithm in nature).

Table 3-(b) shows results gathered for second version where each processor sends

its message directly to its recipient. Nearly in all combinations Converse is faster than PVM. From the point of EPX, Converse is slow especially for combinations including small message sizes and small number of processors. But as message size and number of processors increase, Converse's performance increases. Since Converse is built on top of EPX, for sending a message in Converse some steps/functions should be passed/called until the final EPX call on CMI layer. This cost becomes high especially for small message sizes, which avoids exploiting the property of message-driven execution.

6 Conclusion

In this paper, we described an efficient layer for porting Converse to Parsytec CC distributed memory machine. Although, we ported Converse on top of Parsytec's native message passing layer (EPX), the performance results show that Converse performs equally with native message passing on simple (synchronous communication) algorithms and outperforms on algorithms that involves asynchronous communication. With this implementation, other parallel languages such as Charm++, a parallel object oriented language, and libraries built with Converse have become available to be used on the Parsytec machine. We have already being developing a parallel-object oriented application (hierarchical parallel radiosity [11] using Charm++. If Converse could be implemented on top of hardware using (instead of EPX layer) (which is also a future work for us) we believe it will perform even better.

References

[1] *Converse Programming Manual*, Department of Computer Science, University of Illinois at Urbana-Campaign, available at `http://charm.cs.uiuc.edu`, Nov. 1996.

[2] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, Active Messages: A Mechanism for Integrated Communication and Computation, *Proc. 19th Ann. Int'l Symp. Computer Architectures, ACM Press, New York*, pp. 256-266, 1992.

[3] *Embedded Parix, Software Documentation.*

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM 3 User's Guide and Refence Manual*, OAK Ridge National Laboratory, Knoxville, TN, Technical Report ORNL/TM-12187, Sep. 1994.

[5] L. V. Kale, M. Bhandarkar, N. Jagathesan, and S. Krishnan, Converse: An Interoperable Framework for Parallel Programming, *Proc. of IPPS'96*, pp. 212-217, 1996.

[6] K. Langendoen, R. Bhoedjang, and H. Bal, Models for Asynchronous Message Handling, *IEEE Concurrency*, April-June 97, pp. 28-38, 1997.

[7] S. Pakin, M. Lauria, and A. Chien, High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet, *Proc. Supercomputing'95 (CD-ROM), CS Press, Los Alamitos, California*, 1995.

[8] L. V. Kale and A. Gursoy, Modularity, Reuse and Efficiency with Message-Driven Libraries, *Proc. of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pp. 738-743, 1995.

[9] L. V. Kale and S. Krishnan, Charm++: A Concurrent Object-Oriented System Based on C++, *Proc. OOPSLA'93*, pp. 91-108, 1993.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing, 1994.

[11] A. R. Sireli, Parallel Hierarchical Radiosity, M.S. Thesis (in progress), Bilkent University, Ankara, 1998.

[12] A. Gursoy, Simplified Expression of Message-Driven Programs and Its Impact on Performance, Ph.D. thesis, University of Illinois at Urbana-Campaign, May 1994.

[13] M. Bhandarkar and L.V. Kale, "MICE:A Prototype MPI Implementation in Converse Environment, *Proc. of the second MPI Developers Conf.*, pp. 26-31, South Bend, Indiana, July 1996.