A Portable Run-Time System for Object-Parallel Systems

Peter Beckman
Dennis Gannon

CRPC-TR95607
November 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

A Portable Run-Time System for Object-Parallel Systems *

Peter Beckman, Dennis Gannon

Department of Computer Science

Indiana University, Bloomington, Indiana, U.S.A.

November 8, 1995

Abstract

This paper describes a parallel run-time system (RTS) that is used as part of the pC++ parallel programming language. The RTS has been implemented on a variety of scalable, MPP computers including the IBM SP2, Intel Paragon, Meiko CS2, SGI Power Challenge, and Cray T3D. This system differs from other data-parallel RTS implementations; it is designed to support the operations from object-parallel programming that require remote member function execution and load and store operations on remote data. The implementation is designed to provide the thinnest possible layer atop the vendor-supplied machine interface. That thin veneer can then be used by other run-time layers to build machine independent class libraries, compiler back ends, and more

*This research is supported in part by: ARPA DABT63-94-C-0029 and Rome Labs Contract AF 30602-92-C-0135

sophisticated run-time support. Some preliminary measurements of the RTS are given for the IBM SP2, SGI Power Challenge, and Cray T3D.

1 Introduction

Object Parallelism is a term which we will use to describe a family of related approaches to programming parallel computers. This model extends data parallelism to object-oriented languages and systems. More formally, object parallelism is defined by the concurrent application of parallel functions and operators to aggregate data structures. In an object-parallel system, the programmer creates and manipulates *collection* or *container* objects that encapsulate a distributed data structure on a massively parallel computer. Examples of this style of programming include

1. pC++ [1], a collection based parallel programming language.
2. Single Program Multiple Data (SPMD) C++ class libraries that provide user defined, encapsulated support for parallel, distributed data structures. Examples include LPARX[2], P++[3], the POOMA library[4] and CHAOS++[5].
3. Data-parallel extensions to the C++ Standard Templates Library such as the Amelia Vector Template Library (AVTL)[6].

While C++ is the most common language for this work, it is not the only way object parallelism may be expressed. There are many other parallel programming models that exploit object-oriented concepts. These include Mentat[7], CC++[8], CORBA[9], Charm++[10], and UC++[11]. In these systems, the emphasis is on task-level parallelism on networked

and heterogeneous systems. As will be explained later, the run-time architectures for these systems have different requirements than for the data-parallel extensions addressed here.

In this paper, we describe a run-time system designed to support object parallelism on scalable, massively parallel processing (MPP) computers. It has been implemented on a variety of scalable, MPP computers including the IBM SP2, Intel Paragon, Meiko CS2, SGI Power Challenge, and Cray T3D. The design of the system was governed by a set of objectives and requirements that were derived from our experience with the pC++ project. We were motivated by the following concerns.

1. The RTS will serve as a *compiler target* as well as an API for parallel class library designers. Consequently, the RTS design should be guided by optimization rather than end user's programming ease.

2. Because the current standard practice for writing parallel class libraries is to use SPMD style, the RTS should be based on this model. However, it should also provide a path to a more general *multi-threaded* implementation style. In particular, future support for dynamic, nested data parallelism and mixed task-data parallelism will require multi-threaded implementations.

3. Basic message passing should be a fundamental part of the RTS and the MPI standard [12] is now well supported on many systems. However, object parallelism differs from data parallelism. Unlike uniform arrays, distributed complex linked structures change size and shape over the lifetime of the program. Consequently, translating all data movements into predetermined send-receive pairs is not always easy. Furthermore, object-oriented style lends itself more naturally to a single-sided communication model

where a member function is invoked through a pointer to an object. Active messages [13] and remote procedure call methods must be a component of the RTS.

4. The RTS should provide a consistent interface across all the target platforms, but it should provide explicit support for hardware features that have been added to many parallel systems. A primary difference between an MPP architecture and a network of workstations is the special hardware for collective communication, barriers, and shared memory. The RTS should include primitives such as barrier and remote memory load and store, whose implementation can use special hardware when it is available.

The work described here addresses three of these four concerns. This paper does not consider the interface to multi-threaded execution. A working group, known as POrtable Run Time System (PORTS) [14], is currently considering the design of a multi-threaded implementation. We also emphasize that we have not considered issues related to nested data parallelism, which may be handled by a multi-threaded system or by compiler technology such as that pioneered in the design the NESL system [15].

In addition, a complete RTS would address the issue of interoperability between parallel class libraries, parallel programming languages like pC++, and data-parallel languages like HPF. Such a RTS should provide a foundation for a low-level distributed data structure library that can be shared by many language and library implementations. A second consortium of researchers, the Parallel Compiler Run Time Consortium (PCRC) [16], is considering this problem. The RTS design presented here is based, in part, on requirements for that effort.

Our focus is on the design problems associated with building a run-time system that supports collective operations, remote data load and store, and remote procedure calls (RPC) within an SPMD execution environment. It will be shown that these goals are often conflicting. For example, fast hardware barriers are often not implemented in a manner that supports the asynchronous nature of RPC operations. However, it will be shown that it is still possible to design a RTS that builds upon the hardware primitives and operates with "reasonable efficiency".

We are not the first to address these problems. The pioneering work on *active messages* provides a powerful example of how these problems can be addressed. The active message system is based on a limited form of remote procedure call, i.e., messages are sent between processors consisting of a message handler and a few bytes of argument data. The message size is bounded by the architecture of the processor network interface adaptor. Messages can either interrupt the addressed processor, or a remote-queue and polling mechanism can be used. Our approach differs in that we allow arbitrary argument lengths, and a single (complex) RPC handler per processor.

NEXUS [17] is an example of a run-time system that is designed for multi-threaded task parallelism on heterogeneous networks. Communication in NEXUS is based on an RPC mechanism and there is no attempt to exploit special hardware for collective communication or remote shared memory. Internally, NEXUS uses the thread interface specified by the PORTS consortium. The future multi-threaded implementation of the RTS described here will share that layer. Of course, the most common run-time layer for SPMD C++ parallel class libraries is MPI. It is also used by the Amelia parallel vector template library. For the library writer, the RTS described here can extend the functionality of MPI with support for

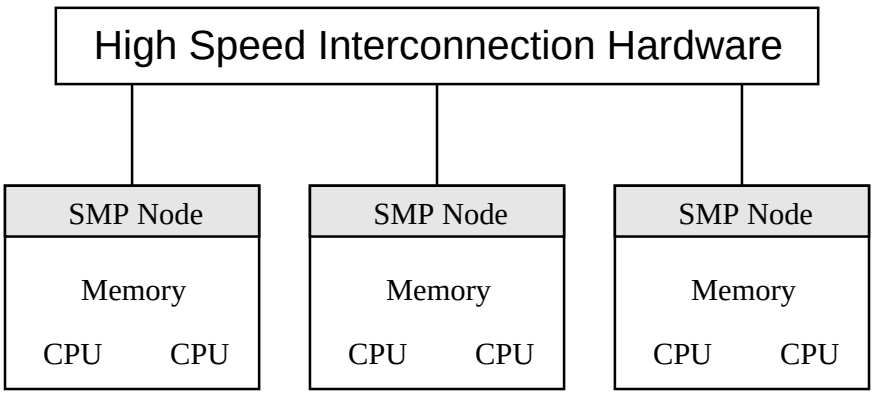

Figure 1: The Virtual Parallel Machine

one-sided communication and remote load and store operations for those times when it is not possible or convenient to use matched send/receive communication.

In the paragraphs that follow, we will describe the abstract machine model, the interface to the software, and the implementation. We conclude with a description of a series of experiments.

2 The Abstract Machine

Object-parallel systems built atop this RTS are designed to run on a wide range of architectures. The run-time system uses an abstract machine model that incorporates the high performance machines in production today, as well as the future directions of high performance architecture design. Figure 1 shows a block diagram for the virtual machine used in the run-time system. The following definitions provide vocabulary for discussing the RTS view of the underlying hardware:

SMP: Symmetric multiprocessor. An SMP is a basic building block; it contains one or more CPUs that can share memory and have cache coherency. The operating system

provided interface to concurrency may be in the form of kernel-level threads, virtual processors, or allocation of physical CPU resources. Most workstation vendors provide high-end SMP servers with 2-8 processors.

Node: An SMP, possibly connected to other SMPs via a high speed network. A *NodeNumber* uniquely identifies each node.

Context: An address space. A Unix process on a SMP would be a single context. Lightweight threads share a context. A machine such as the SP2 can support several contexts per node. A *ContextNumber* is used to uniquely identify each Context.

3 Three High Performance Machines

To explore the design and implementation issues for the RTS, three very different machines were selected, and the low-level RTS interface ported to each of them. Below, is a quick tour of each of the computers selected for the prototype.

3.1 Distributed Memory: The IBM SP2

The IBM SP2 is a traditional distributed memory message passing supercomputer. However, unlike the TMC CM5, Intel Paragon, and Cray T3D, each compute node is a complete computer, with it's own disk and host name. IBM produces several POWER2 microprocessor based nodes for the SP2. However at the present time, SP2 compute nodes have only one processor. This makes mapping the SP2 to the parallel virtual machine quite simple - each processor is one node.

IBM interconnects the SP2 nodes with a multi-stage high performance switch (HPS)

capable of 40 MB/sec bandwidth and application to application latency of 40 microseconds. While there are several software interfaces for communication on the SP2, we chose an experimental IBM implementation [18] of MPI installed on the SP2 at the Cornell Theory Center for this version of the run-time system.

3.2 Network DMA: The Cray T3D

The T3D is a distributed memory 3D torus-connected compute array of DEC Alpha microprocessors. For the T3D at the Pittsburgh Supercomputer Center, each processing element (PE) is a 150 Mhz Alpha with 64 megabytes of local memory. Two PEs share each high speed interconnection network node, which has a peak performance of 300 MB/sec along each link. A convenient interface to the interconnection hardware is Cray's C communication library. The RTS uses the library, which provides "get" and "put" functions for moving remote memory without interrupting the remote node (network DMA).

3.3 Shared Memory: The SGI Power Challenge

The 75 Mhz. MIPS superscalar R8000 microprocessor chipset is the heart of SGI's Power Challenge (SGIPC). The SGIPC at the Indiana University Computer Science Department has 2 GB of total memory and 10 processors, each equipped with a 4 MB data cache. The SGIPC uses snooping cache coherency hardware and the high speed bus can sustain a total peak transfer rate of 1.2 GB/sec.

SGI compilers are designed to interface the hardware support for *piggyback reads*, which can combine read requests so that other processors can share the response, and *synchro-*

nization counters, which count broadcast increment transactions and help improve barrier synchronization times. However, the software interface for RTS designers is clumsy. There is no standardized thread interface, such as Pthreads, so users are stuck with programming `SPROC`s, a multiprocessor version of `fork()`. Furthermore, there is no easy way to insist that an `SPROC` execute on a particular compute node. In true SMP style, the individual tasks are moved from processor to processor based on unseen heuristics within the operating system.

3.4 Communication Abstractions

The object-parallel execution model is based on the parallel invocation of class member functions for a distributed set of objects. Many operations use pointers to objects that may exist within the same context or in a remote context. To support this concept we introduce a global name space defined in terms of a generalization of pointers. Like NEXUS, we define two types of pointers.

LocalPointer: A simple, untyped, memory address. A LocalPointer is valid only within the Context it was created.

GlobalPointer: The tuple (ContextNumber, LocalPointer). A GlobalPointer uniquely identifies any memory address in the computational hardware.

The abstract services based on these pointer types are remote procedure calls and remote memory fetch/store operations.

4 Run-Time Functionality

The RTS provides functions for remote memory load/store, remote procedure invocation, collective operations, timers, process control, and tracing and profiling. In the subsections that follow we discuss the first three of these. We will not discuss timers, process control, or tracing and profiling in this paper.

4.1 Remote Memory Operations

The table below shows the interface to the global pointer communication routines. FetchStart() and StoreStart() are very much like `memcpy()`. For FetchStart(), the source address is a global pointer, and the destination is a local pointer. With StoreStart() it is switched. Since the functions are non-blocking, the address to a *service handle* must be passed into both functions. Later, the handle may be checked to see if the operation has completed. Probe() and Wait() behave as expected, and can be used to test or block on a service handle.

Also in the table below is Barrier(), a collective synchronization function. The RTS supports other collective functions such as reduce and broadcast, but they are beyond the scope of this paper.

4.2 Remote Procedure Invocation

Even on a shared memory machine such as the SGIPC, writing a remote procedure call mechanism can be difficult. There are two ways to detect the arrival of a RPC request, Poll() for it, or rely on an interrupt mechanism. An interrupt or active message mechanism embodies the asynchronous nature of an RPC nicely. However, standard Unix signals must

Function Name	Description
`void FetchStart(...)`	Begin to fetch data from remote source to local destination. A service handle can be tested for completion
`void StoreStart(...)`	Begin to store data from local source to remote destination. A service handle can be tested for completion
`void Wait(...)`	Wait on service handle until preposted service call completes. `Poll()` is called while waiting
`int Probe(...)`	Return condition of service handle. Has the preposted service call completed?
`void Barrier()`	Barrier synchronization

Table 1: Basic Remote Memory Functions

Function Name	Description
`void RemoteActionHandler(...)`	The function called to decode and execute a remote invocation request.
`void RemoteAction(...)`	Ask a remote node to execute `RemoteActionHandler()` with the provided buffer. A service handle can be tested for completion
`void Poll()`	Check and service any pending events

Table 2: Remote Action Functions

go through many operating system layers, and usually have high latency. Our experiments revealed that a program running on two processors of the SGIPC takes about 33 milliseconds of wall clock time to ping-pong a Unix signal between them. We decided to poll for the arrival of RPC requests rather than pay that latency. Since the SGIPC has snooping caches, polling for a RPC request is very fast, and is usually a few simple cache loads. Deciding when to poll can be difficult. Setting a timer is one solution. The pC++ compiler takes a different tack, it "sprinkles" calls to Poll() into the code. Periodic calls to Poll() were required on all the computers.

Rather than provide a way to "register" function names and pointers, we chose an interface only a compiler could love. There is only one handler for each node. That handler

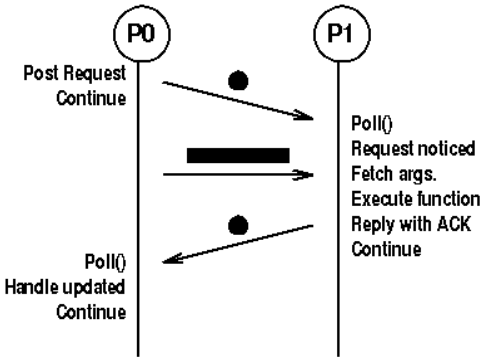

Figure 2: RemoteAction() on the T3D

is written by the compiler, or by the library writer. When a node issues a *RemoteAction* request, a message is sent to the receiver, and queued. The receiver must execute Poll() to check for its arrival. After the receiver notices the request, it executes RemoteActionHandler(). RemoteAction() needs only three parameters, a ContextNumber where the handler should be run, an integer type field, and a buffer (usually used to encode procedure arguments). The type field is an easy way provide the RemoteActionHandler() with information about the buffer.

We are currently working on the compiler transformations required to convert global pointer member function invocations into a call to RemoteAction(). The compiler would write the code required to pack and unpack the member function arguments in and out of the buffer.

5 Implementation Details

5.1 RemoteAction()

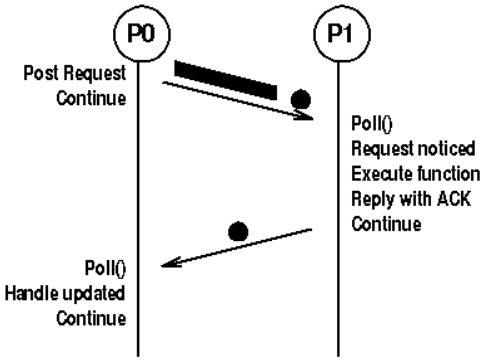

Figure 3: RemoteAction() on the SP2

Figure 2 shows the interaction between two T3D nodes communicating via remote procedure call. In the figure, the message type is encoded as shape. A short control message is drawn as a dark circle, while a large data block is a rectangle.

The T3D can fetch the argument buffer directly from the remote node. This eliminates a buffer copy and some extra synchronization protocol. Shared memory machines can skip the explicit fetch of the RPC argument buffer and simply use the pointer to the buffer. Unfortunately, a space leak results unless the original buffer is freed after RemoteActionHandler() exits. On most shared memory machines, the memory can be freed by the node executing the handler. However on the T3D, only the **owner** can free the buffer, and the buffer may not be freed until the handler completes. One solution is to post an ACK. In the same way Poll() checks for RPC events, it would check for ACK receipts. After detecting an ACK, the buffer could be freed and the service handle updated.

Figure 3 shows the implementation of RPC semantics over a message passing layer. Notice that the control message and data message are combined. This eliminates a round trip to set up matched send and receive pairs by forcing the messaging layer to buffer and hold the message until Poll() notices their arrival and posts a RECV() to pull them in from

the system buffers. Since argument buffers are not large, this generally not a problem.

After the handler exits, an ACK control message is sent to the caller. A Poll() will detect the control message, and the corresponding service handle will be modified.

5.2 Barrier()

The RTS for a node cannot block indefinitely. This would suspend user-level threads and prevent the node from servicing any RemoteAction() requests. Unfortunately most vendor-supplied barrier synchronization functions are blocking. Call their barrier function, and control is not returned until all processors enter the barrier and synchronize. We cannot use those types of vendor supplied barriers. For the SGIPC and SP2, we were forced to write our own barrier, which could execute Poll() and service requests while waiting at the barrier. Cray saw the wisdom of a providing a non-blocking barrier function interface on the T3D. Calling `set_barrier()` simply sets a bit on the hardware and returns immediately. Processors can then execute `test_barrier()` and Poll() until the barrier is over.

On the SP2 and SGIPC, a simple fan-in/fan-out binary tree was constructed to provide barrier. Nodes entering the barrier notify their parent node they are waiting at the barrier. When both all a node's children have so registered, the event is propagated. While the nodes wait for the fan-out message, they execute Poll(), and may service other requests. When the root node receives a message from its children, the barrier is done, and notification fans out in the reverse direction.

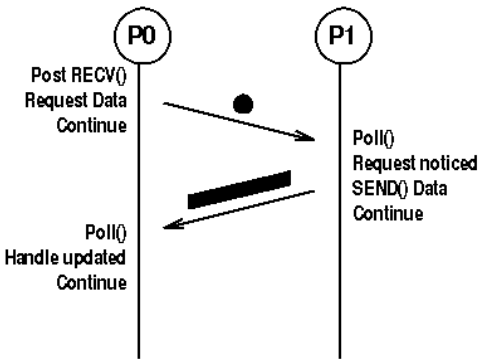

Figure 4: FetchStart() on the SP2

5.3 FetchStart()

FetchStart() provides a "receiver driven" communication primitive. On shared memory machines, FetchStart() should never be called, since the compiler should be smart enough to generate code for shared memory. Nevertheless, if used, it is basically a call to `memcpy()`. For network DMA machines such as the T3D, FetchStart() calls the appropriate memory transfer function. On the T3D, this was complicated by a block transfer engine that operates on 8 byte words. Doing a FetchStart() when the source and destination are misaligned can force extra network reads, masking, and buffering.

Figure 4 depicts the rendezvous protocol required to efficiently implement this type of unscheduled communication on the SP2. Unneeded buffer copies – especially for large messages, **must** be avoided. If sender and receiver carefully choose a message type, and the receive is preposted, extra buffer copies are eliminated. Assuming that both sender and receiver have agreed on a message type, Figure 4 shows how a FetchStart() can be initiated, and the computation resumed. Notice where a multi- threaded system could overlap computation and communication.

To agree on a message type to match up each send/receive pairs requires the message `type`

field be split into two logical fields: a message ID (MID) and a message flavor. This technique is quite similar to the way *Chant* [19] uses MPI to send messages to threads, except in this case, the extra bits stolen from the message type field are not used to encode destination, but to distinguish between pending FetchStart() messages. Each pending FetchStart() must have a unique message ID, so the RTS can match senders and receivers, and detect which preposted receives have completed, and which corresponding service handles should be updated.

5.4 StoreStart()

StoreStart() is very similar to FetchStart(). On shared memory machines, StoreStart() should be avoided; no explicit RTS calls are needed to move data. On the T3D and other network DMA machines, StoreStart() calls the vendor-supplied data transfer engine. StoreStart() is not atomic. Unpredictable results occur when simultaneous writes overlap.

The current implementation of StoreStart() is not quite satisfactory on the T3D. The word-aligned transfer engine forces at least 8 bytes on the remote node to be modified. If the destination is not word aligned, a read must first pull out the existing data, combine it with the bytes that should be modified, and write the entire word back out to the remote node. This creates a race condition. If two variables share the same word, a StoreStart() to one of them will also read and then write the other. Intermediate changes to the variable would be lost. Fortunately, all variables and heap-allocated memory segments are word aligned. Nevertheless, a better solution must be found.

StoreStart() on the SP2 is also more complex that its FetchStart() counterpart. As Figure 5 indicates, to avoid extra buffer copies, an extra message must be ping-ponged

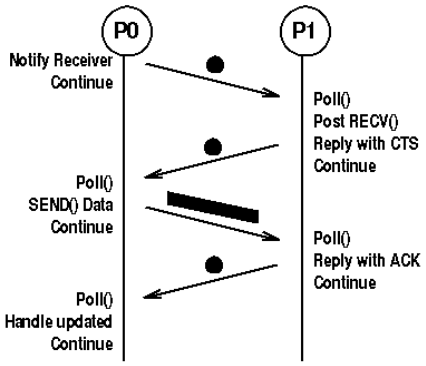

Figure 5: StoreStart() on the SP2

between source and destination. The first message announces the intent of the sender. The receiver (P1) must post a RECV() in anticipation of the data. A control message is returned to the sender indicating it is clear to send (CTS). After the receiver detects that the message has indeed arrived, it sends an ACK to the sender, to notify the node the transaction is complete.

6 Early Performance Results

6.1 Communication

In this section we present some of the early performance tests of the run-time system. Figure 6 compares FetchStart() on the SP2 and T3D with the native communication layer provided by the vendor. Remember that FetchStart used "one sided" communication. On the SP2, the remote "owner" of the data simply executes Poll(). When a control message arrives, it decodes and services it. For the T3D, the owner of the data Polls(), but since network DMA communication occurs without owner participation, the owner remains idle.

For the native SP2 communication test, raw synchronous MPI_Ssend() and MPI_Recv()

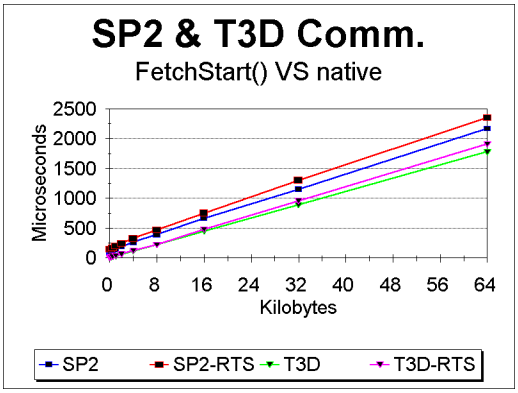

Figure 6: FetchStart() on the SP2 and T3D

SP2	IBM SP2 using MPI
SP2-RTS	IBM SP2 with Run-Time System
T3D	Cray T3D using native `shmem_get()`
T3D-RTS	Cray T3D using Run-Time System

between two nodes was used. On the T3D, a call to shmem_get() was all that was necessary. For a 64K transfer, the T3D native layer achieved about 35 MB/sec bandwidth, while MPI on the SP2 got about 31.5 MB/sec.

Figure 6 indicates that a small, nearly constant overhead was associated with using FetchStart() instead of the native communication layer. Because of the current design of Poll(), the added latency varies with the number of processors. Clearly, Poll() should get first priority as the RTS is optimized and tuned.

6.2 RemoteAction

In this test, a RemoteAction() request was sent to a node, which in turn, sent a RemoteAction() event back. This ping-pong was repeated, and the time averaged. For the SP2, with nodes waiting at the barrier and polling, the ping-pong RemoteAction() averaged about 165 microseconds. On the T3D, the time was 87.3. Dividing these numbers in two, gives the

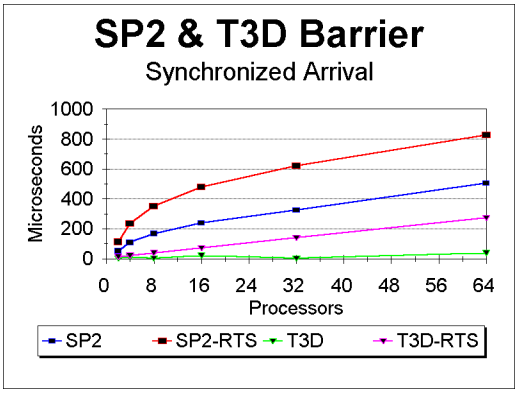

Figure 7: Synchronized Arrival

SP2	IBM SP2 using native `MPI_Barrier()`
SP2-RTS	IBM SP2 with Run-Time System polling barrier
T3D	Cray T3D using native `barrier()`
T3D-RTS	Cray T3D using Run-Time polling barrier

optimal one-way RemoteAction() latency at 82.5 and 43.7 microseconds for the SP2 and T3D respectively.

6.3 SP2 and T3D Barrier Synchronization

Barrier synchronization on the run-time system must be non-blocking, and execute Poll() between tests for completion. Figures 7, 8, and 9 show the performance of the vendor-supplied barrier against the RTS barrier for the SP2 and T3D. The times shown represent the average wall clock time spent in the barrier

The first test shows the performance of barrier when the nodes are nearly synchronized. For this case, the barrier was repeated over and over, with only a single floating point operation between calls. Of course this is not a realistic window into how user code performs. However it does provide an interesting look at how the barrier was designed, and if there is extra contention when nodes arrive together.

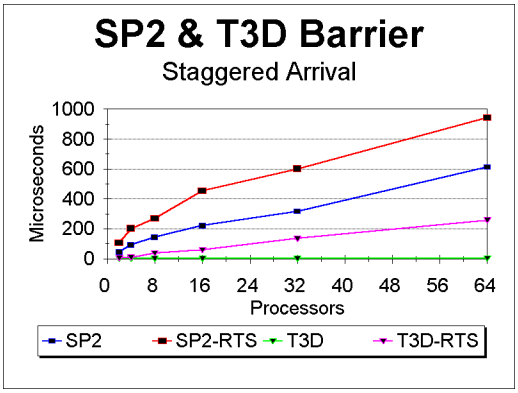

Figure 8: Staggered Arrival

Figure 7 has several interesting features. First, is the blazing fast T3D native barrier. In all of our tests, from 2 to 64 nodes, the barrier was never slower than 4 microseconds. Cray Research uses a series of wires and fast logic to make their hardware-assisted barrier. This is a perfect example of why the RTS **must**, whenever possible, use the fast hardware provided by the vendor. The performance of the RTS barrier for the T3D once again demonstrates that the delays added by Poll() should be minimized. For the SP2, the RTS polling barrier was within a factor of about 2 to 3 from the MPI blocking barrier.

A more likely barrier scenario for real programs is represented by Figure 8. For this case, each node arrived at a slightly different time. After averaging over many arrival sequences and nodes, the results show that staggered arrival is a small improvement over synchronous arrival.

Figure 9 shows the average time for a barrier when all the other nodes are waiting, and one node arrives late. For this test, the SP2 times varied widely. When all the other nodes have already entered the barrier, the late arrival of the root node immediately triggers a fan-out message that the barrier is over. Half of the barrier is therefore already complete when the root arrives. On the other hand, if a leaf in the barrier tree arrives late, notification

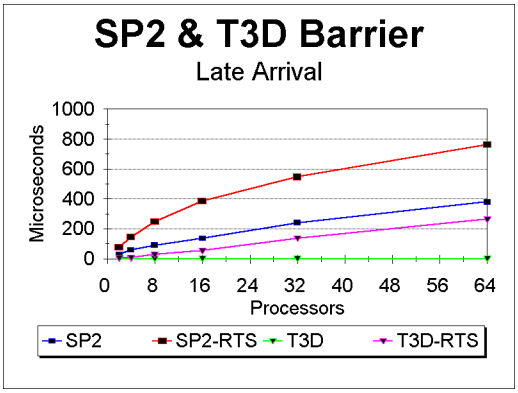

Figure 9: Late Arrival

SGI-Sync	Native barrier with synchronous arrival
RTS-Sync	RTS polling barrier with synchronous arrival
SGI-Irreg	Native barrier with staggered arrival
RTS-Irreg	RTS polling barrier with staggered arrival
SGI-Late	Native barrier with late arrival
RTS-Late	RTS polling barrier with late arrival

must trickle up and down the height of the tree. There was often a factor of 2 or more between the fastest late arrival barrier time (the root) and the slowest (a leaf). For example, with 64 nodes on the SP2, the late arrival MPI barrier times varied between 303 and 839 microseconds, depending on which node was the latecomer (and recorded the time).

6.4 SGI Power Challenge Barrier Synchronization

The RTS barrier implementation for the SGIPC proved to be the biggest surprise. In the first implementation, the tree barrier, had linear performance; the tree was not a tree. Careful investigation revealed that our entire reduction tree was within a single 128 byte cache line – serializing our updates. A careful distribution of tree nodes improved performance. In some cases, our polling barrier outperformed the native SGI barrier.

The same three barrier experiments were repeated on the SGIPC. The results are dis-

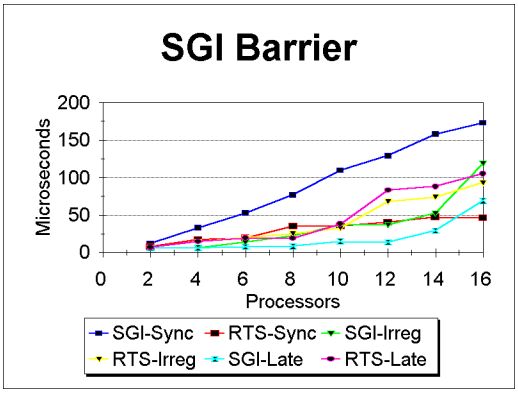

Figure 10: SGI Barrier

played in Figure 10. Remember that the SGIPC is a SMP, and the virtual SPMD nodes of the RTS migrate around the machine. Timings for the SGIPC were varied. Future benchmark tests should be done with the machine in single-user mode. The slowest times were for the vendor-supplied barrier running the synchronous arrival test. The near-linear performance suggests that their implementation does not use a tree, but rather all the nodes fight for control over some resource. The more nodes, the more contention. For most cases, the tree barrier of the RTS performed much like the SGIPC barrier.

7 Conclusions

In this paper we have described a simple portable, light-weight run-time system for scalable MPP architectures. The design emphasizes the types of operations that are important for implementing object-parallel extensions to the data-parallel model. This RTS incorporates, remote memory fetch and store operations, remote procedure call, and collective communication and synchronization. The design of the primitives in this system are intended for use by a compiler or a parallel library designer. Consequently, they are to be as fast and efficient

as possible. In particular, they are designed to take advantage of any architectural feature that the host parallel computer provides.

In our experiments we measured the performance of this design on three different machines. For the SP2, we built RPC and memory fetch/store operations on top of MPI. Adding those features has a cost; barrier synchronization is more complex, and about twice as slow. The cost of fetch/store operations is nearly the same as MPI synchronized send/receive pairs. For the CRAY T3D, we used the existing hardware barrier to build an extended barrier that allows asynchronous communication (RPCs) to take place while processors wait. Our initial implementation is not very fast compared to the 4 microsecond hardware barrier, but it is still faster than the native MPI barrier on the SP2. Furthermore, we feel that the costs can be substantially reduced by a carefully tuned, low-level implementation.

One conclusion that can be drawn from this work is that an extension of the MPI standard to include active messages, RPC operations and remote memory load and store operations, would allow vendors to optimize these operations to match their special architectural features. Consequently, library designers and compiler writers would be given much greater opportunity to optimize performance for a given machine than they currently have.

8 Bibliography

1. D. Gannon, F. Bodin, P. Beckman, S. Yang, and S. Narayana. *Distributed pC++: Basic ideas for an object parallel language*, Journal of Scientific Programming, Vol. 2, pp. 7-22 (1993).

2. Scott B. Baden, Scott R. Kohn, Silvia M. Figueria, and Stephen J. Fink, *The LPARX*

User's Guide v1.0, Technical report, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 USA, Apr. 1994.

3. R. Parsons, D. Quinlan, *A++/P++ Array Classes for Architecture Independent Finite Difference Computations*, Proceedings, OONSKI, 1994. pp. 408-418.

4. J. Reynders, D. Forslund, P. Hinker, M. Tholburn, D. Kilman, W. Humphrey, *Object Oriented Particle Simulation on Parallel Computers*, Proceedings, OONSKI, 1994. pp. 266-279.

5. C. Chang, A. Sussman, J. Saltz, *Support for Distributed Dynamic Data Structures in C++*, Univ. of Maryland, Dept. of Computer Science Technical Report CS-TR-3266, 1995.

6. T. Sheffler, *A Portable MPI-based parallel vector template library*, RIACS Technical Report 95.04, RIACS, NASA Ames Research Center. 1995.

7. A. S. Grimshaw. *An introduction to parallel object-oriented programming with Mentat*, Technical Report 91 07, University of Virginia, 1991.

8. K. M. Chandy and C. F. Kesselman. *CC++: A declarative concurrent object-oriented programming notation*, In Research Directions in Object-Oriented Programming, MIT Press, 1993.

9. CORBA 2.0 specification available through OMG. See WWW reference below

10. L. Kale, S. Krisnan, *Charm++: A Portable Concurrent Object Oriented System Based on C++*. Technical Report. Univ. of Illinois, Urbana-Champaign, 1994.

11. UC++ - Europa Parallel C++ Project. WWW reference only: http://www.lpac.qmw.ac.uk/europa/.

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

13. von Eicken, Thorsten; Culler, David E.; Goldstein, Seth Copen; Schauser, Klaus Erik. *Active Messages: a Mechanism for Integrated Communication and Computation.* UCB//CSD-92-675, March 1992. 20 pages.

14. PORTS- POrtable Runtime System consortium. *The PORTS0 Interface*, Technical Report, Jan. 1995.

15. Guy E. Blelloch, *NESL: A Nested Data-Parallel Language* CMU CS Technical Report CMU-CS-92-103.ps January 1992 36 pages

16. PCRC- Parallel Compiler Runtime Consortium. Private communication. see http://www.extreme.indiana.edu/pcrc/index.html.

17. Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke *Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems* Technical Memorandum ANL/MCS-TM-189, May 1994 TM189.dvi.Z TM189.ps.Z

18. Hubertus Franke, *MPI-F An MPI Implementation for the IBM SP-1/SP-2*, Version 1.39, internal document, IBM T.J. Watson Research Center, Feb. 95,

19. M. Haines, D. Cronk, P. Mehrotra, *On the design of Chant: A talking threads package.* Proceedings of Supercomputing 1994, Washington, D.C. Nov. 1994.