Addressing Communication Latency Issues on Clusters for Fine Grained Asynchronous Applications - A Case Study*

Umesh Kumar V. Rajasekaran, Malolan Chetlur,
Girindra D. Sharma, Radharamanan Radhakrishnan,
and *Philip A. Wilsey*

Computer Architecture Design Laboratory,
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221–0030
Ph:(513) 556-4779, Fax:(513) 556-7326, phil.wilsey@uc.edu

Abstract. With the advent of cheap and powerful hardware for workstations and networks, a new cluster-based architecture for parallel processing applications has been envisioned. However, fine-grained asynchronous applications that communicate frequently are not the ideal candidates for such architectures because of their high latency communication costs. Hence, designers of fine-grained parallel applications on clusters are faced with the problem of reducing the high communication latency in such architectures. Depending on what kind of resources are available, the communication latency can be improved along the following dimensions: (a) reducing network latency by employing a higher performance network hardware (i.e., Fast Ethernet versus Myrinet); (b) reducing communication software overhead by developing more efficient communication libraries (MPICH versus TCPMPL (our TCP/IP based message passing layer) versus MPI-BIP); (c) rewriting/restructuring the application code for less frequent communication; and (d) exploiting application characteristics by deploying communication optimizations that exploit the application's inherent communication characteristics. This paper discusses our experiences with building a communication subsystem on a cluster of workstations for a fine-grained asynchronous application (a Time Warp synchronized discrete-event simulator). Specifically, our efforts in reducing the communication latency along three of the four aforementioned dimensions are detailed and discussed. In addition, performance results from an in-depth empirical evaluation of the communication subsystem are reported in the paper.

1 Introduction

One of the primary bottlenecks limiting the performance of distributed applications is the communication latency. The cost of communication operations is significantly higher than computation operations because of the overhead involved

* Support for this work was provided in part by the Advanced Research Projects Agency under contracts J–FBI–93–116 and DABT63–96–C–0055.

in preparing a message and the electrical delay necessary for signal processing across bandwidth-limited physical network links. The nature of the communication interface in a message passing environment is such that the software overhead (time required for the preparation and authentication of the message) is significantly higher than the hardware overhead (network setup and message propagation time) [5]. Under such circumstances, an important lesson that has been learned is to minimize the frequency of communication, but not necessarily the size of the messages in order to arrive at an efficient implementation. However, a large class of distributed applications tend to have fine-grained asynchronous communication characteristics. This implies that it is not always possible to minimize the frequency of communication and other methods are needed to alleviate the overhead of message latency.

Several strategies for minimizing the communication latencies of distributed applications have been explored. Depending on what kind of cost-performance trade-offs are required, the communication latency can be improved along the following four dimensions: (a) using faster network hardware; (b) using more efficient communication libraries; (c) rewriting or restructuring the application code for less frequent communication; and (d) using communication optimizations that exploit the application's inherent communication characteristics. In this paper, we discuss our experiences with building a communication subsystem on a cluster of three workstations (dual processor Pentium Pros running Linux) for a fine-grained asynchronous application (namely WARPED, a Time Warp synchronized discrete-event simulator). Specifically, our efforts in reducing the communication latency along three of the four aforementioned dimensions are detailed and discussed. As rewriting and restructuring the application code for less frequent communication is relatively a well studied and application-specific method [5], we will not discuss it here in this paper. However, each of the other three dimensions are dealt in detail and supported by empirical analysis. The remainder of this paper is organized as follows. Section 2 overviews parallel discrete event simulation (which is our domain of interest) and describes the experimental environment. Section 3 discusses our experiences with new network hardware (Myrinet) and presents a comparison between different network hardware configurations and their effects on the application. Section 4 illustrates the effect of different communication libraries on the performance of the application and presents a comparison between MPICH, MPI-BIP and TCPMPL (our native TCP/IP based message passing library). Section 5 discusses the effect of two communication optimizations (message aggregation and infrequent polling) that have been developed to further improve the performance of the communication subsystem. Finally, Section 6 presents some concluding remarks.

2 Background

In this section, a brief overview of a class of fine-grained asynchronous applications that is of interest to us is presented. The field of parallel discrete event simulation (PDES) [6] contains several representative examples of fine-grained

asynchronous applications. In PDES, the model to be simulated is decomposed into *physical processes* that are implemented as *simulation objects*. Each simulation object is assigned to a *Logical Process* (LP); the simulator is composed of a set of LPs concurrently executing their simulation objects. Simulation objects communicate by exchanging time-stamped messages through the LPs. Thus, each LP (which can be associated with multiple simulation objects) receives messages from other LPs and forwards them to the destination objects. In order to maintain causality, LPs must process messages in strictly non-decreasing time-stamp order [10]. There are two basic synchronization protocols used to ensure that this condition is not violated: (i) *conservative* and (ii) *optimistic*. Conservative protocols [13] avoid causality errors, while optimistic protocols, such as Time Warp [6, 10] allow causality errors to occur, but implement some recovery mechanism.

In a Time Warp simulator such as WARPED [17], each LP operates as a distinct discrete event simulator, maintaining input and output event lists, a state queue, and a local simulation time (called *Local Virtual Time* or LVT). As each LP simulates asynchronously, it is possible for an LP to receive an event from the past (some LPs will be processing faster than others and hence will have local virtual times greater than others) — violating the causality constraints of the events in the simulation. Such messages are called *straggler* messages. On receipt of a straggler message, the LP must rollback to undo some work that has been done. The state and output queue are present to support rollback processing. Rollback involves two steps: (i) restoring the state to a time preceding the time-stamp of the straggler and (ii) canceling any output event messages that were erroneously sent (by sending *anti-messages*). After a rollback, events are re-executed in the proper order.

2.1 The Experimental Environment

The WARPED simulation kernel provides the functionality to develop applications modeled as discrete event simulations [17]. Considerable effort has been made to define a standard programming interface to hide the details of Time Warp from the Application Programming Interface (API). All Time Warp specific activities such as state saving and rollback are performed by the kernel without intervention from the application. Consequently, an implementation of the WARPED interface can be constructed using either conservative [13] or optimistic [6] parallel synchronization techniques. Furthermore, the simulation kernel can operate as a sequential kernel. A more detailed description of the internal structure and organization of the WARPED kernel is available on the www at `http://www.ececs.uc.edu/~paw/warped`.

The results reported in this paper were obtained by executing four different simulation models on a cluster of three Pentium Pro workstations interconnected by 100Mbps Fast Ethernet and 1.28Gbps Myrinet. To fully test the system, the cluster of workstations chosen for the experiments were not dedicated to the experiments. Five sets of measurements were taken at two different times and

the average of these values were then reported. The average granularity[1] (in terms of computation time) of each application is as follows: RAID (20μsecs), SMMP (15μsecs), QUEUE (41μsecs), and PHOLD (20 μsecs). The four models (available in the WARPED distribution) are:

RAID: The RAID application models the RAID Disk Arrays which is a method of providing a vast array of storage [2] with a higher I/O performance than several large expensive disks. This application incorporates a flexible model of a RAID Disk Array and can be configured in various sizes of disk arrays and request generators. Each request is in fact a token that carries information about the number of disks, the number of cylinders, number of tracks, number of sectors, size of each sector and specific information about which stripe to read and parity information.

SMMP: The SMMP application models a shared memory multiprocessor. Each processor is assumed to have a local cache with access to a common global memory (The model is somewhat contrived in that requests to the memory are not serialized — *i.e.,* main memory can have multiple requests pending at any given moment). The model is generated by a C++ program which lets the user adjust the following parameters: (i) the number of processors/caches to simulate, (ii) the number of LPs to generate, (iii) the speed of cache, (iv) the speed of main memory, and (v) the cache hit ratio. The generation program partitions the model to take advantage of the fast intra-LP communication.

QUEUE: The QUEUE application is a reconfigurable queuing library with the essential components needed to model queuing systems. The queuing library consists of *source*, *queue*, *server* and *statistics collector* objects. The model and the various parameters of the source, server and queue objects can be set by the modeler through simple configuration files. Different scenarios can be modeled by specifying the desired layout in the configuration file.

PHOLD: The PHOLD model is the parallel hold model first reported by Fujimoto [7]. PHOLD is a synthetic workload model that contains several parameters that are used to test the performance of a parallel simulation. The PHOLD workload model contains the following parameters: number of logical processes, message population, timestamp increment function, movement function, computation grain and an initial configuration.

3 Using Network Hardware to Reduce Network Latency

A cluster of workstations (COW) used for parallel applications consists of low-cost workstations interconnected by a network. Applications running on multiple workstations need a message passing library to communicate information between its peers running on different workstations. The message passing library in turn requires an interconnection network for communicating information across

[1] The granularity of the application is the basic unity of computation. It represents the time taken by a simulation process to execute one simulation cycle.

the workstations. This interconnection network can be shared for many purposes or can be used exclusively for parallel applications. Ethernet and Fast Ethernet are one class of interconnection networks that use the ubiquitous Ethernet protocol. The effective utilization of the Ethernet depends on the number of workstations sharing it in shared network. However, the advent of switched Ethernet solutions have helped in improving the utilization of the network.

Alternatively, the Ethernet network can be replaced by a high performance interconnection network hardware to improve the network latency and the effective bandwidth associated with the network. One such high performance interconnection network hardware is Myrinet [1]. Myrinet is a new type of local-area network (LAN) based on the technology used for packet communication and switching on "massively parallel processors" (MPPs). The following are the features of Myrinet which make it a high performance interconnection network suitable for parallel applications: (a) high bandwidth (1.28Gbps), (b) low latency, and (c) host interface that can handle packet traffic. These features of Myrinet help in increasing the communication subsystem performance. In addition, the processor on the Myrinet control board can share (or take control of) the message processing with the main processor. Since Myrinet provides robust communication channels with flow control, packet framing, and several other features, a message passing library using Myrinet for communication directly from the user level would lead to a high performance communication subsystem. There have been several reports comparing the performance of Fast Ethernet clusters with Myrinet clusters [11, 12].

4 Using Efficient Message Passing Libraries

One of the factors that increased the popularity of workstation clusters is the standardization of tools and utilities needed to run distributed applications on workstation clusters. One such standard is the Message Passing Interface (MPI) [8] standard. MPI defines the interface and semantics of a message passing library. Our fine-grained asynchronous parallel simulations use MPICH for communication on workstation clusters. Due to their inherent fine-grained nature, these simulations communicate very frequently during their execution. There are three factors in a message passing software that decide the performance of these applications to a large extent. They are as follows: (a) time consumed by the send operation; (b) one way message latency; and (c) time consumed by the probe operation. The time consumed by the send operation steals the processor cycles away from the application. The one way message latency of the software affects the performance of these applications indirectly. Application characteristics such as rollbacks in Time Warp simulators are affected by the one way message latency. The larger the one way message latency, the more the probability of a rollback. Another factor that affects the performance of these applications is the message probe operation. Since these applications are asynchronous, they probe for messages at regular intervals to check for the arrival of messages. On a successful probe operation, the applications receive the message and resume their

work. This section presents the features of MPICH and MPI-BIP. In addition, the following section details how the performance of asynchronous distributed applications can be increased by using efficient message passing libraries.

4.1 MPICH and MPI-BIP

MPICH [9] is a freely available implementation of MPI standard developed by Argonne National Laboratory and Mississippi State University that runs on a wide variety of platforms. The main goals of the architecture of MPICH are: portability and high performance. The design of MPICH was guided by two principles: (a) the maximization of the amount of code that can be shared without compromising performance; and (b) to provide a structure whereby MPICH could be ported to a new platform quickly. Some of the advantages of MPICH are: portability, language bindings for C, Fortran and C++, high performance, heterogeneity and interoperability.

We considered two alternatives in providing an efficient message passing layer to our applications. One way was to have a high cost, high performance interconnection network (such as Myrinet) together with a high-performance message passing layer to take advantage of the faster network hardware. One example of such a message passing layer is the Basic Interface for Parallelism (BIP) [16] protocol. There are other fast messaging layers that provide user-level network access. Some of them are U-Net [19], FM [15], GAMMA [4], and PARMA^2 [12]. While some of these libraries operate over Myrinet, some operate over Fast Ethernet. BIP is a new protocol designed for high performance networking on Myrinet. Prylli *et al* [16] have designed and implemented BIP such that it exploits the high speed Myrinet network to the fullest, without spending time in system calls or memory copies, giving all the bandwidth (5 μs message latency) to the applications using it. They have also designed and implemented MPI-BIP, a MPI interface on top of BIP to support the portability of parallel applications adhering to the MPI interface. The MPI interface of BIP was one of the primary reasons for us choosing MPI-BIP. This significantly reduced the programming costs involved in using the new message passing library as all our applications are MPI-based. One disadvantage of the BIP library is the exclusive access to the network by only one application running on a workstation [16]. The BIP implementation monopolizes the network interface of one node for each application. Therefore, no other applications can use the network for communication from the same workstation. The other alternative was to have a low cost, reasonable performance interconnection network (such as Fast Ethernet) and to develop a message passing library similar to MPICH but with reduced functionality to increase the performance of the library. This motivated the development of TCPMPL, a TCP/IP based message passing library.

4.2 TCPMPL

The other dimension in which effort (in terms of programming costs) was spent in improving the performance of the communications subsystem, was in the de-

velopment of a custom message passing library (TCPMPL) over TCP/IP with minimal functionalities. Clearly we did not require the overhead of the extensive library support of MPICH to run our distributed applications. Some of the features of MPICH which we considered as being unnecessary for our purposes are as follows:

- *Heterogeneous Workstations*: MPICH is designed to be run on heterogeneous workstations. Our asynchronous applications run only on homogeneous workstations and therefore, do not require this overhead.
- *Functionalities*: The MPI standard specifies a myriad of functions to encompass all the communication operations performed by parallel applications. Our asynchronous applications use only a limited set of functions in the MPI standard such as send, receive, and probe.
- *Platform Independence*: MPICH has been designed to be portable to multiple hardware platforms. This feature is not necessary if the software is not intended for different hardware platforms.

Therefore, limiting a message passing library to provide only the functionality required by our asynchronous applications has the potential of increasing the communication subsystem's performance. One of the main goals for TCPMPL is portability and high performance. The C++ class interface of TCPMPL is given below:

```
class TCPMPL {
      TCPMPL(void);
      ~TCPMPL();
      void  initTCPMPL(int &argc, char ** &argv);
      int   getId(void) const;
      int   getTCPMPLSize()() const;
      bool  sendData(char *ptr, int nbytes, int id);
      bool  sendDataVector(const struct iovec *vector, int count, int id);
      bool  recvDataFrom(char *ptr, int nbytes, int id);
      bool  probeForData(int *source, int *length, long seconds, long microseconds);
}
```

The following are the minimal requirements of TCPMPL which are required by our asynchronous applications for correct functional execution:

- *SPMD*: TCPMPL must support the Single Program Multiple Data (SPMD) concept. Applications compiled with the library should have the feature of starting the parallel applications on different workstations and synchronize among themselves before executing the application code. This functionality is accomplished by the *initTCPMPL* method.
- *Message Delivery*: The service provided by TCPMPL must be reliable and must ensure ordered delivery. TCPMPL should reliably deliver the messages to the destination process. It should also preserve the ordering (FIFO) of the messages from a particular process.
- *Message Boundary*: TCPMPL should preserve the message boundary, as the applications use it to send messages.

– *Send*, *Probe*, and *Receive*: Application should be able to send a message to any particular process in the group of processes started by TCPMPL. This is handled by the *sendData* and *sendDataVector* communication methods. Since our applications are asynchronous by nature, support for polling for messages that can be received without blocking is required. This is handled by the *probeForData* method. Application processes should be able to receive messages from any other process. This is taken care of by the *recvDataFrom* communication primitive.

One of the foremost design decisions was the choice of the underlying protocol to be used in TCPMPL. There were two choices: (a) TCP - a connection oriented protocol that provides reliability and ordered delivery; and (b) UDP [18] - a connectionless datagram protocol that does not guarantee reliability and ordered delivery. Since our applications require reliable message transmission and ordered delivery, we chose TCP instead of UDP as TCPMPL's underlying protocol. Since the TCP implementation is available as one of the operating system services on most platforms, our library would also be portable to most platforms. Establishment of communication channels for communicating information between the processes involves different design choices. One approach is to establish communication channels between all the processes so that all communication channels are setup before the application starts to execute. A disadvantage of this approach is that not all processes in the group will communicate with each other during their entire execution. The advantage of this approach is that all the communication channels are setup ahead, so there is no overhead during the execution of the application for establishing communication channels. Another design approach would be to establish communication as and when the processes communicate. Resources can be utilized more efficiently by using this approach. The disadvantage is that the control cannot be given to the message passing layer at both the ends for establishment of communication channels since this involves some protocol specific activity which has to be synchronized at both ends. Another approach to minimizing the communication channels would be to have a spanning tree of the processes in which the edges denote the bi-directional communication channels and the nodes represent the processes. This design would decrease the usage of resources but would increase the one way message latency. We wanted TCPMPL to have the minimum one way message latency and lower overhead during the execution of the application. Hence, we chose to have all the communication channels established before the application starts to execute.

In addition, TCPMPL has to preserve message boundaries while delivering a message to the user application. For example, if an user application sends two messages of size s_1 and s_2 bytes, TCPMPL should deliver the first message of size s_1 bytes followed by the message of s_2 bytes to the user. TCPMPL should not deliver the first s_1+s_2 bytes as a single message or in any other combination. To handle message boundaries, TCPMPL sends a message header containing the size of the message that is going to follow the message header. TCPMPL has a primitive error detection mechanism to detect any errors caused by TCP

in delivering the messages to the destinations. It sends the sender's id and the destination's id in the message header to detect any errors in the delivery of the messages by TCP. The base version of TCPMPL performed well and certain optimizations were incorporated to further improve the performance of TCPMPL. Some of the optimizations are discussed below:

Reduction in socket I/O: TCPMPL sends a message header for every message that is being sent by the user application. In the base version, TCPMPL sends the message header followed by the user message. Since each write to the socket involves a switch from user-mode to kernel-mode, these calls (one for sending the message header and another call to send the user message) result in a lot of overhead. Hence, both calls were merged to reduce the switching overhead, so that both the writes are performed in one call.

Reduction in copying: When a user application wants to send two buffers together as a single message, the application has to copy the two buffers into a single message and then send it using TCPMPL. To avoid the extra copying, an additional interface to TCPMPL was provided that takes multiple buffers and sends them as a single message. This avoids the extra copying.

Disabling Nagle's algorithm: The size of TCP's header is 20 bytes and IP requires 20 bytes for its header. Therefore, for small message sizes, this 40 byte header wastes the bandwidth of the network. To avoid this, Nagle's algorithm [14] is used in TCP to delay sending of small messages when there is any outstanding segment of data that has not been acknowledged. More data can be accumulated until the acknowledgment for the outstanding packet has arrived. The accumulated data is then sent as a single packet. There is also a delayed acknowledgment timer in TCP which delays the acknowledgment of data being sent so that it can piggy-back with the data in the opposite direction. Nagle's algorithm and delay acknowledgment timer together added a significant delay to the packets. This increases the one way message latency of TCPMPL as the majority of messages transmitted by our fine-grained asynchronous applications were of the order of a few hundred bytes. To avoid this delay, TCPMPL disables Nagle's algorithm throughout the application execution.

Socket Buffer Size: TCP provides both reliability and ordered delivery to the higher layers in the TCP/IP protocol stack. TCP has internal buffers for buffering the user data. Since it assumes the underlying network is not reliable, it buffers the data and retransmits it if the destination process has not received the data. There are buffers at the sending end for this retransmission purposes. There is also a buffer at the receiving end to buffer the data before the application can receive the data. Both sender and receiver buffer sizes were increased to 65536 bytes to increase the performance of TCPMPL. If the buffer size is small, TCP will block until it gets an empty buffer to buffer the data.

Table 1. Round trip times & bandwidth of MPICH, TCPMPL and MPI-BIP

Packet Size	Round Trip Time			Bandwidth		
	MPICH	TCPMPL	MPIBIP	MPICH	TCPMPL	MPIBIP
bytes	μ secs	μ secs	μ secs	Mbps	Mbps	Mbps
128	627	440	38	3.27	4.65	53.89
256	676	560	104	6.06	7.31	39.38
512	868	797	114	9.44	10.28	71.86
1024	1354	1271	124	12.10	12.89	132.13
2048	1977	1893	149	16.57	17.31	219.92
4096	2814	2565	197	23.29	25.55	332.67

4.3 Performance of TCPMPL and MPI-BIP over MPICH

Figure 1 illustrates the performance of four parallel, asynchronous WARPED applications with MPICH, TCPMPL and MPI-BIP. From the figure, it is clear that all applications run with TCPMPL perform much better than the same applications run with MPICH. By tailoring the message passing library to suit our needs, we were able to get on average, a 75% improvement on all the applications tested. These improvements were on a cluster of three workstations (dual processor Pentium Pro) interconnected with Fast Ethernet. Table 1 illustrates the raw performance numbers (round trip times and bandwidth) for all three communication libraries. Specifically, we present the round-trip times and bandwidth of TCPMPL and MPICH on a Fast Ethernet network while round-trip times and bandwidth of MPI-BIP were collected on a Myrinet network.

The experiments on the Myrinet platform were carried out to study the effect of lower message latencies on the fine-grained asynchronous applications. To address this issue, we experimented with a Myrinet network interconnecting the same cluster of workstations. In addition, we chose to use a message passing library that took advantage of the faster underlying network (MPI-BIP). It turns out that MPI-BIP on Myrinet performs about 10 times faster than MPICH on Fast Ethernet. This translates to a further 40% improvement on average over the TCPMPL on Fast Ethernet combination, on all the applications that were tested. Figure 1 shows the same four WARPED applications performing even better with the MPI-BIP on Myrinet combination.

5 Using Communication Optimizations in the Application Layer

So far, we have established that faster hardware and efficient libraries reduce the communication latency. However, the communication latency experienced by distributed applications is inevitable. Cautious exploitation of communication characteristics of applications can further reduce the average communication overhead. For fine-grained asynchronous applications running on clusters, even a small communication latency greatly affects the performance. This fact calls

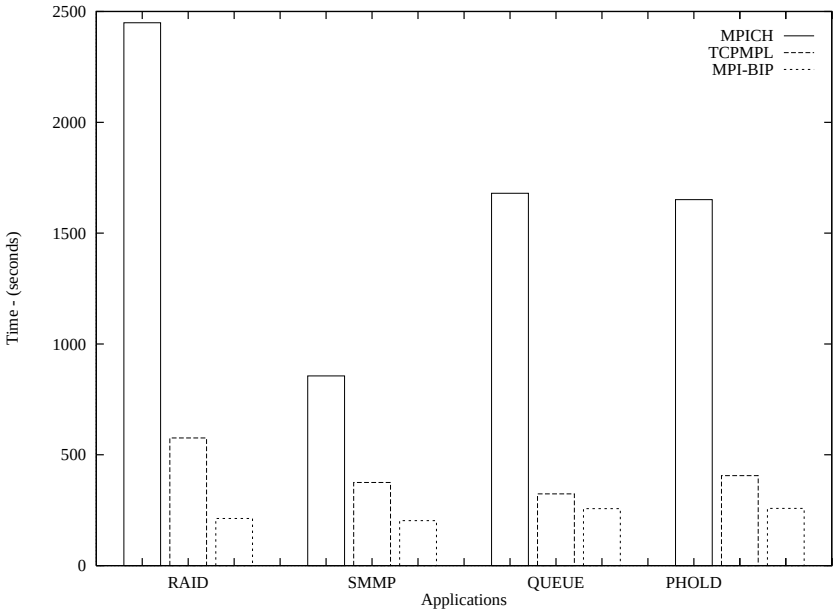

Fig. 1. Effect of message passing libraries and hardware on performance

for optimizations in the application layer that could exploit the communication characteristics to reduce the communication overhead.

Table 2. Communication costs for 128 byte messages

Library	Probe		Receive	Send
	unsuccessful (μ secs)	successful (μ secs)	(μ secs)	(μ secs)
MPICH	22	298	16	101
TCPMPL	21	63	17	44
MPI-BIP	1	12	5	15

In this section, we discuss two optimizations namely *message aggregation* and *infrequent polling* that can be employed in the communication module of fine-grained asynchronous applications. The performance analysis of various message passing libraries showed that communication send operations incur a fixed cost irrespective of the message size. Although timely reception of these messages is desirable in asynchronous applications, there is a local computation that can proceed independent of the reception of these messages. Therefore, by aggregating several application messages into a single physical message, we only incur the communication overhead for the one physical message instead of incurring the

overhead for every application message sent over the network. This will clearly benefit fine-grained asynchronous applications. Similarly, the processes must poll the network to determine whether any new messages have been received. Applications poll frequently to receive messages as soon as they arrive, but at the cost of increased overhead. Polling infrequently to reduce the polling overhead without affecting the progress of simulation improves performance. Table 2 illustrates the send, receive and probe overheads (in terms of time) for messages of size 128 bytes of all three communication libraries used in this study (MPICH, TCPMPL and MPI-BIP). In the following subsections, we will discuss in detail these two optimizations and their impact on the communication behavior of asynchronous applications.

5.1 Message Aggregation

Message Aggregation (MA) is an optimization to the Time Warp communication subsystem that matches the communication behavior to the underlying communication subsystem. Using MA, the communication module of each LP collects application messages destined to the same LP, that occur in close temporal proximity, and sends them as a single physical message. Since there is a large overhead associated with each message (regardless of the message size), significant improvement in performance is possible. The decision on when to send the messages is made by the *aggregation policy*.

Clearly, the higher the number of messages aggregated, the greater the reduction in the communication overhead. The longer the messages are delayed, the greater the number of messages aggregated. However, delaying messages excessively harms the performance of the receiving LP. Thus, aggregation policies must balance the potential gain from additional message aggregation, to the potential loss at the receiving LP. It is difficult to determine a static balance between the two factors; the communication behavior of Time Warp simulators is highly dynamic and unpredictable. While static window sizes (in time, or number of messages) for aggregation yield some performance improvement, better overall performance results from dynamic control of the aggregate size.

In specifying the window size, we seek to balance the benefits resulting from aggregating more messages, versus the harm from delaying messages excessively. These two factors are modeled as: (i) Aggregation Optimistic Factor (AOF): AOF is the gain from delaying the messages; it is proportional to the rate of reception of messages to be sent. If AOF is high, a large number of messages are aggregated without excessive delay; and (ii) Aggregation Pessimistic Factor (APF): APF models the harm from delaying the messages. It is proportional to the age of the aggregate. Note that both of these factors vary with the nature of the application, and may change dynamically within the lifetime of the simulation.

Initially, a static policy that aggregates messages for a fixed time, called *Fixed Aggregation Window* (FAW), was developed. The age of the first message received by the aggregation layer is tracked. Once this age reaches a constant value (the size of the window), the aggregate message is sent. The advantage

Table 3. Average Aggregate Size

Application	Msg. Size	# Appl. Messages	# Physical Messages	Avg. Aggregate Size
RAID	132	5133307	3052662	1.68
SMMP	92	977310	870314	1.12
QUEUE	96	1511043	1253779	1.20
PHOLD	92	5413578	320090	1.69

of this policy is its low overhead; only a single check of the current aggregate age (time that the aggregate has been alive) against the constant window size is required. This policy provides a static balance between AOF and APF, making it insensitive to changes in the communication behavior of the application. No matter how high (or low) the message arrival rate is, the fixed window size is used. A fixed window size of 20 (heuristically determined) was used for all experiments with message aggregation. Table 3 illustrates the average aggregate size for all four WARPED examples. Since the chosen window size significantly affects the performance of this policy, dynamic control of the balance between AOF and APF is being investigated [3].

5.2 Infrequent Polling

In asynchronous applications, the process polls the communication layer for messages, receives the messages if any are available, performs some computation and sends some messages. If the message probe operation successfully discovers a message, the message is received, and the network is polled again for messages until no more messages are detected. The optimization is to carry out the message polling periodically instead of polling for messages after every computation. The benefit is that there is a lower chance that the polling is unsuccessful since it is performed infrequently and there is a longer period of time available for messages to arrive between polls.

The analysis of the performance of asynchronous distributed applications that use polling demonstrates that it is possible to enhance the performance by optimizing the message delivery, especially for fine-grained applications. Unfortunately, developing the criteria for optimal polling frequency even under approximate conditions is difficult. Even if it was possible to directly compute the optimal polling frequency, it is likely to vary during the lifetime of the application, and from process to process. Therefore, we suggest a infrequent polling strategy that arrives at the optimal polling frequency heuristically. In this strategy, an initial period for polling is selected at compile time, and enforced on all the processes. The advantage of this policy is that it adds no overhead. For the experiments conducted, each application preferred a unique polling frequency. The following polling frequencies[2] were used for experiments conducted on the Fast Ethernet network: (a) RAID - preferred a polling frequency of 3; (b) SMMP

[2] A polling frequency of n implies that a probe is conducted after processing n events.

- preferred a polling frequency of 25; (c) QUEUE - preferred a polling frequency of 10; and (d) PHOLD - preferred a polling frequency of 15. The same polling frequencies were used for TCPMPL and MPICH based experiments as there wasn't any appreciable difference in the unsuccessful polling costs. However, the polling costs for MPI-BIP are much lower than the Fast Ethernet polling costs. So for experiments conducted with MPI-BIP, all applications preferred a polling frequency of 3.

5.3 Performance of Message Aggregation and Infrequent Polling

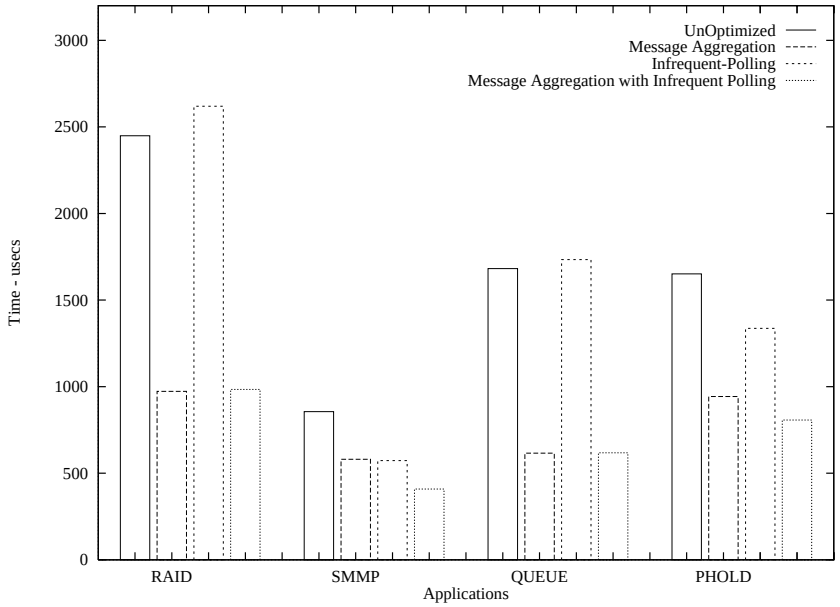

Fig. 2. Effect of optimizations on performance using MPICH on Fast Ethernet

In order to study the effect of message aggregation and infrequent polling on fine-grained asynchronous applications, four WARPED applications were tested with different message passing libraries. Figures 2, 3, and 4 illustrate the effect of the optimizations on the performance of the applications when executed with MPICH, TCPMPL and MPI-BIP respectively.

Figure 2 validates our assertion that aggregation can be beneficial to fine-grained asynchronous applications when there is a significant latency in message transmissions. As seen in the figure, message aggregation (using the MPICH message passing library on a Fast Ethernet network) provides on average 50% performance improvement (over the unoptimized version) on all the applications tested. This improvement can be attributed to the reduction in communication

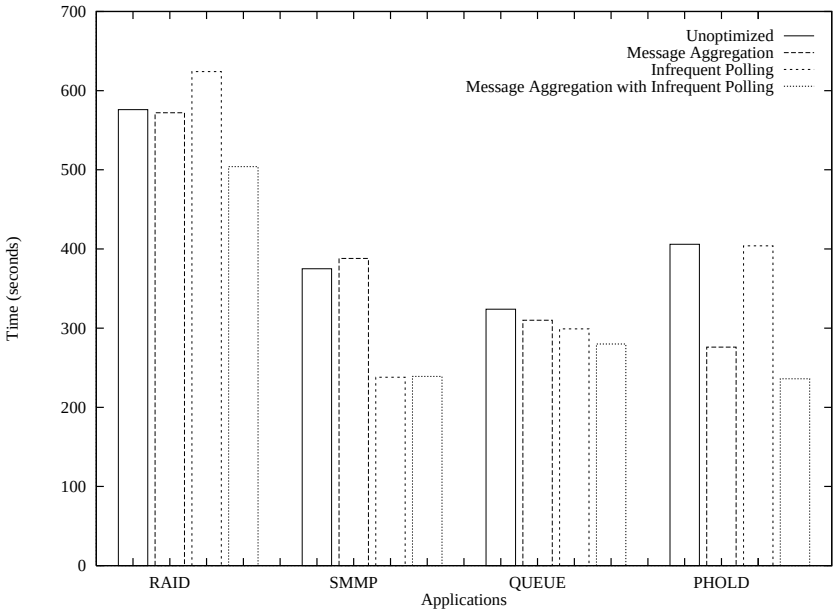

Fig. 3. Effect of optimizations on performance using TCPMPL on Fast Ethernet

overhead as a lower number of physical messages are sent across the network. On the other hand, the infrequent polling optimization does not fare well with the applications tested. The majority of the applications tested (excluding the SMMP model) generate large amounts of messages and infrequently polling for messages actually degrades the performance of the applications. However, when the message aggregation optimization is deployed along with the infrequent polling strategy, there is a significant improvement in performance (57%, *i.e.*, an improvement of 7% over the speedup obtained by having message aggregation alone). This improvement can be attributed to the fact that the message aggregation optimization reduces the number of messages sent across the network and in this scenario, the infrequent polling strategy is beneficial to the application.

Figure 3 illustrates the effect of the optimizations on the performance of the applications when using TCPMPL on a Fast Ethernet network. As TCPMPL significantly reduces the software overhead involved in message transmission, message aggregation does not perform as well as it did with MPICH. As messages are sent with lower latency, deployment of the message aggregation optimization results in only a 8% improvement on average on all the applications tested. The infrequent polling optimization performs much better here than it did with MPICH, mainly because messages are communicated faster and there are occasions where there are no message available in the channel. Barring one application (RAID), infrequent polling provides a small performance improvement for all the other applications tested. So it comes as no surprise that when these two optimizations are combined together, a further performance improve-

ment is obtained. A performance improvement of 26% was obtained when both optimizations were deployed.

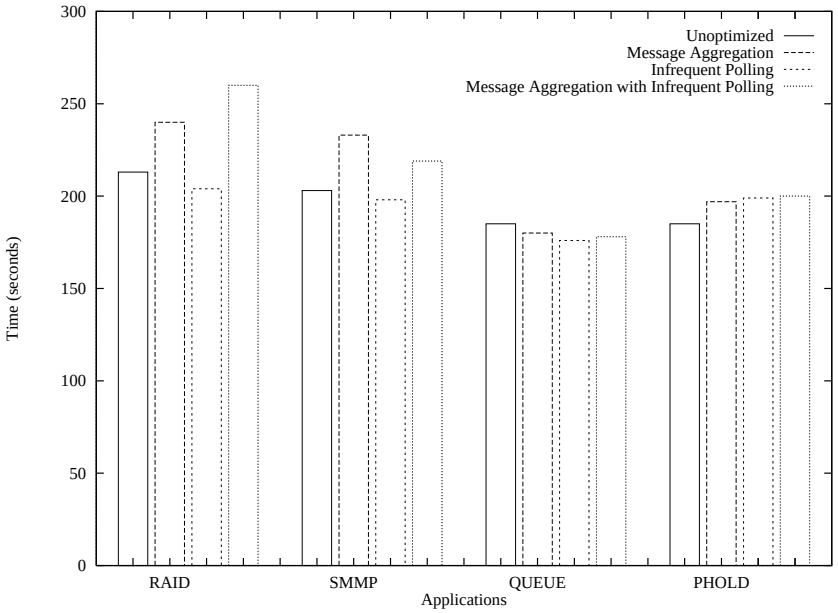

Fig. 4. Effect of optimizations on performance using MPI-BIP on Myrinet

Finally, the effect of the optimizations on the performance of the applications when using the MPI-BIP message passing library on a Myrinet network is illustrated in Figure 4. From the figure, it is clear that on a fast network with a fast message passing library, the optimizations do not improve the performance of the application. Barring a slight performance improvement with infrequent polling, the optimizations (alone and combined) do not significantly improve the performance of the applications. This was expected as the optimizations took advantage of inefficiencies of the Fast Ethernet network and the message passing libraries to improve the performance of the applications. When these inefficiencies were removed (by adding a faster network layer and a fast message passing library), the optimizations were ineffective.

6 Conclusions

The performance of fine-grained asynchronous applications on clusters is directly influenced by the underlying communication subsystem. There is a wide latitude in the choice of hardware and software components for building the communication subsystem. Faster communication networks with efficient communication libraries definitely improve the performance of communication operations. This

is clearly demonstrated by the results. We also see that the communication libraries play a major role in the effective utilization of the underlying network bandwidth. This is illustrated by the TCPMPL library which performs better than the MPICH implementation. From this we can conclude that a message passing library with minimal functionality and implementation optimizations such as (i) reduced socket I/O, (ii) reduced copying, and (iii) suitable buffer sizes can significantly improve the performance of fine-grained asynchronous applications. In addition, adhering to the standard interfaces enables the portability of the distributed applications. We also see that, in addition to improving the underlying network and communication libraries, exploiting the application's communication behavior can improve the execution performance. An intelligent combination of these three components in any communication subsystem will significantly improve the performance of a distributed application.

It is clear from the empirical analysis that, even with a relatively inexpensive network (such as Fast Ethernet), improvement in performance of fine-grained applications can be achieved when time (and programming effort) is spent on improving the message passing library and developing application-based communication optimizations. As seen from the figures, TCPMPL along with the two communication optimizations gives performance that is comparable (both in terms of cost and performance) to the MPI-BIP on Myrinet combination. In addition, the time spent in developing a message passing library is justified if one considers the cost of the expensive Myrinet hardware. In this paper, we presented our efforts in finding viable solutions that provide good performance on low-cost hardware technology. In addition, to really appreciate the performance improvement, we compared the Fast Ethernet results with performance results obtained from the same cluster interconnected with the Myrinet network and which uses an optimized message passing library (MPI-BIP).

We also observed that the communication optimizations (message aggregation and infrequent polling) were not able to exploit the application's behavior in high speed networks such as Myrinet. Currently, work is in progress to develop optimizations that can detect application communication characteristics that can be exploited in high speed networks. Clearly there is a wide variety of design choices in building a low cost communication subsystem. The design choices should be made with the cost (money and time) and performance in mind. High speed networks definitely reduce communication latency. However the intelligent combination of the underlying network hardware and efficient message passing libraries coupled with algorithmic optimizations can make COWs a suitable environment for fine-grained asynchronous applications.

References

1. Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. Myrinet – a gigabit-per-second local-area network. *IEEE Micro 15*, 1 (February 1995), 29–36.
2. Chen, P. M. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (June 1994), 145–185.

3. Chetlur, M., Abu-Ghazaleh, N., Radhakrishnan, R., and Wilsey, P. A. Optimizing communication in Time-Warp simulators. In *12th Workshop on Parallel and Distributed Simulation* (May 1998), Society for Computer Simulation, pp. 64–71.
4. Ciaccio, G. Optimal communication performance on fast ethernet with gamma. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing* (Orlando, Florida, March/April 1998), Springer, pp. 534–548.
5. Felten, E. W. Protocol compilation: High-performance communication for parallel programs. Tech. rep., University of Washington — Dept. of Computer Science, 1993.
6. Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (Oct. 1990), 30–53.
7. Fujimoto, R. Performance of time warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (Jan. 1990), 23–28.
8. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M. *MPI: The Complete Reference Volume 2 - The MPI-2 Extension.* MIT Press, 1998.
9. Gropp, W., Lusk, E., Doss, N., and Skjellum, A. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, July 1996.
10. Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.
11. Lab, S. C. Scl cluster cookbook - technology comparison. (available on the www at `http://www.scl.ameslab.gov/Projects/ClusterCookbook/icperf.html`).
12. Marenzoni, P., Rimassa, G., Vignali, M., Bertozzi, M., Conte, G., and Rossi, P. An operating system support to low-overhead communications in NOW clusters. In *Proceedings of Communication and Architectural Support for Net work-Based Parallel Computing CANPC97* (San Antonio, Texas, Feb. 1997), vol. 1199, Springer-Verlag, pp. 130–143.
13. Misra, J. Distributed discrete-event simulation. *Computing Surveys 18*, 1 (Mar. 1986), 39–65.
14. Nagle, J. Congestion control in TCP/IP internetworks. *Computer Communications Review 14* (Oct 1984), 11–17.
15. Pakin, S., Lauria, M., and Chien, A. High performance messaging on workstations: Illinois fast message(FM) for Myrinet. In *Proceedings of Supercomputing '95* (December 1995).
16. Prylli, L., and Tourancheau, B. BIP: A new protocol designed for high performance netowrking on myrinet. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing* (Orlando, Florida, March/April 1998), Springer, pp. 472–485.
17. Radhakrishnan, R., Martin, D. E., Chetlur, M., Rao, D. M., and Wilsey, P. A. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. LNCS 1505. Springer-Verlag, Dec. 1998, pp. 13–23.
18. Stevens, W. R. *TCP/IP Illustrated Volume 1: The Protocols.* Addison-Wesley Publishing Company, Reading Massachusetts, March 1996.
19. von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 3-6 1995).