Minimizing Overhead in Parallel Algorithms Through Overlapping Communication/Computation

Arun K. Somani*and Allen M. Sansano
{arun, allen}@shasta.ee.washington.edu
Department of Electrical Engineering
University of Washington, Box 352500
Seattle, WA 98195-2500

Abstract

One of the major goals in the design of parallel processing machines and algorithms is to reduce the effects of the overhead introduced when a given problem is parallelized. A key contributor to overhead is communication time. Many architectures try to reduce this overhead by minimizing the actual time for communication, including latency and bandwidth. Another approach is to hide communication by overlapping it with computation. This paper presents the Proteus parallel computer and its effective use of communication hiding through overlapping communication/computation techniques. These techniques are easily extended for use in compiler support of parallel programming. We also address the complexity or rather simplicity, in achieving complete exchange on the Proteus Machine.

*This research in part was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while this author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

1.1 Background

This paper presents Proteus [12], a parallel architecture designed to take advantage of overlapping communication/computation. Overlapping communication/computation minimizes the effect of communication overhead introduced when a problem is parallelized. The Proteus project was started in 1990 and working hardware was available in early 1993. A 32 (46 including control and spare) processor machine is being used in current work. One novel architectural feature of Proteus is the use of a dedicated processor for communication control. Other architectures utilizing a communication processor and overlapping communication/computation do exist. In the commercial world the Intel Paragon [4] was introduced at about the same time with a dual processor node with one processor acting as a communication processor. Recent node releases for the Paragon include nodes with up to three processors with one of them acting as a communication processor. The MANNA [2] project at GMD in Germany also utilizes 1 communication processor + 1 computation processor per node. The PIM/c [14] machine from Japan has an 8 processor cluster with one communication processor. The FLASH Project [7] [6] at Stanford utilizes the MAGIC communication processor to handle the communication tasks for a given node in the system. Proteus was the first among these machines to utilize multiple processors per communication processor. The result is a node that can take advantage of overlapping communication/computation at lower costs since the communication hardware is shared among processors on a node.

1.2 Contributions

When a problem is parallelized, the time for inter-processor communication, not found in the serial version, is added to the processors' execution time. This communication time is considered overhead. Communication of data between processors is initiated by a kernel call to invoke a communication handling routine. Then, the data needs to be transmitted. Finally, the communication needs to be completed with the sending and receiving data being resolved. Much time is wasted in the management and transmission of the data. Allocating communication resources and attaining a connection contribute to communication latency. Inefficient protocols and hardware are possible causes of longer latency. Also, the communication link can be slow, or worse, not available for a given amount of time due to the link and/or I/O buffer contention. These inefficiencies can lead to the communicating processors laying idle waiting for communication resources and data.

Many techniques such as low contention interconnection networks, low latency communication protocols, fast data links, and efficient problem partitioning and scheduling to minimize communications are employed to reduce communication overhead. These attempts are good in general, but are lacking in the following way. For communication overhead to approach zero, the communication time would have to approach zero as well. This is not a practical or realizable goal. Overlapping communication/computation is an additional technique used to break the serial communication/computation pattern. This method attempts to hide communication overhead through the use of special hardware. This hardware handles the bulk of communication administration and frees the processor to concentrate on computation tasks during the time it would usually be servicing communication functions. Proteus provides the necessary hardware to overlap communication and computation.

This paper is organized in the following fashion. Section 2 describes the Proteus system architecture and discusses uniques features of Proteus. Section 3 discusses the communication protocol of Proteus. Section 4 outlines the idea of overlapping communication/computation and demonstrates its usefulness through some examples on the Proteus Parallel Computer. Section 5 presents a mechanism to achieve a complete exchange on this machine and compares its performance with other commercial machines such as the Intel Paragon, IBM SP2, and Meiko CS-2. In Section 6, we present our conclusions.

2 Proteus System Architecture

The Proteus Parallel Computer [12] is a Multiple Instruction, Multiple Data (MIMD) machine optimized for large granularity tasks such as machine vision and image processing. This hierarchical system employs shared memory Clusters of processors. The Cluster itself is a powerful shared memory multiprocessor. The advantages of a shared memory model as well as the limitations of scalability in a shared memory architecture are well documented. Proteus scales by connecting Clusters together through a crossbar network. For further scalability, Groups of Clusters can be connected together through some interconnection such as an Enhanced Hypercube [3]. Within this hybrid machine a unique communication system was developed based upon both shared memory and message passing principles. This system allows efficient utilization of shared Cluster components such as memory, communication subsystems, and Cluster Controllers while avoiding high contention for these resources. The power of the clustered node is balanced with the richness of the interconnection network. A diagram of the system is shown in Figure 1.

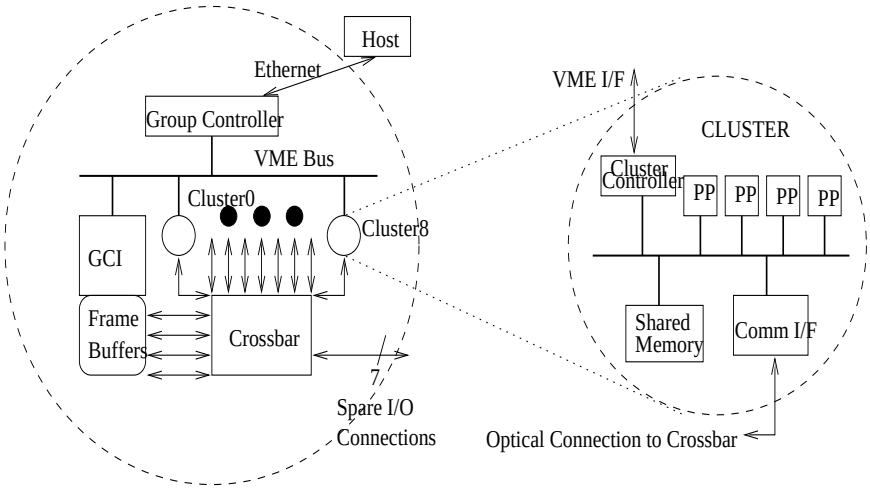

Figure 1: The Proteus System Architecture

Interesting features Proteus include:

- Daughterboard design for the processing element to facilitate upgrading and changing of processors.
- Large one megabyte caches with software controlled, hardware assisted coherency mechanisms at the Cluster level.

- Cache Write Generate caching mode [16].
- Shared communication hardware between processors on a cluster.
- Hardware support for overlapping communication and computation.
- Circuit-switched interconnection with hierarchical control.
- Roving fault tolerance supported by extra hardware.
- Fault tolerant synchronization techniques [5].

2.1 The Pixel Processor

The processing element, or Pixel Processor (PP), used in Proteus is an Intel i860 [17] microprocessor running at 40 MHz. The processor is supported by a one megabyte, unified, direct-mapped L2 cache with a custom cache controller/bus interface at the next level of hierarchy, a shared memory bus. This is the largest amount of cache available with any machine even today. The purpose of the L2 cache is to 1) provide faster memory access and 2) ease the contention on the shared memory bus. The L2 cache will act in various cache accessing modes or as a local memory through an addressing mechanism. The access type is decoded by the cache controller from the upper nibble, 4 bits, of the 32 bit address. This addressing nibble can also specify whether the PP internal cache is also enabled.

Of special interest is the cache generate [16] mode, which has been implemented for research purposes. Cache generate is similar to the write allocate scheme except the line being written into is validated by cache without reading from the main memory. This assumes that the corresponding processor will write into each byte of the cache line before using it for further computation. Hence the reading of a line is not necessary as all data in the line are overwritten. It has been shown that in some cases, cache generate is very beneficial. However, care must be taken to avoid destructive overwriting of data in the cache.

Flushing and invalidation mechanisms are provided for in hardware to handle cache coherency issues. A software command loads a control register in the cache controller signaling it to flush and/or invalidate specified cache lines. Therefore, the programmer (compiler) is responsible for maintaining cache consistency. While this places the burden on the programmer, these techniques reduce costs and complexity related to hardware maintained consistency [8].

In order for the PP to communicate with the Cluster Controller (CC), an interrupt system as well as dedicated mailboxes have been set up. In order to interrupt a CC, the PP first fills in some header type information in a pre-defined shared memory location. Then it sets the appropriate bit in a control register which triggers an interrupt to the CC. The PP is then free to continue on with computations while the CC services the interrupt. A flagging system is set up so that a PP does not corrupt the pre-defined shared memory location on the next interrupt if the previous interrupt has yet to be completed.

Also useful for the PP are some shared memory mailboxes. These are shared memory locations that are dedicated to certain functions between the CC and the PP. For instance, an end-of-transmission (EOT) counter box is set up to keep track of the current EOT count. The EOT is a system clock used to define communication cycles. The CC increments this counter box and the PP reads it for such things as time stamping messages. Other mailboxes are used for the PP to

give information to the CC. For instance, send requests are posted to shared memory mailboxes and are read by the CC. More details on these mechanisms are given in the next section.

2.2 The Cluster

The main components of each cluster are the PPs, a Cluster Controller (CC), and a communication interface.

The Cluster Controller: The CC is primarily responsible for managing all the hardware resources within that cluster. It is an Intel i960 microprocessor. The CC is attached to the shared memory bus and also has it's own local bus. One megabyte of local RAM is provided on the local bus. The CC's tasks include:

- Scheduling tasks on the PEs.
- Managing the shared and dual port (DP) memory resources.
- Managing cluster communication through the optical links.
- Handling all interrupts to the Cluster.
- Acting as PP's interface to the outside world setting up I/O like operations from the shared memory to the VME bus through the Group Controller and Group Host.
- Loading PP code and bringing the PPs out of reset.
- Interfacing with the GCI to coordinate crossbar communication.
- Handling almost all communication functions so the PP does not have to.

The Communication Interface: Each cluster is equipped with an input and an output optical, serial link. Each is set to run at 250 megabits/second, although they can be set up to run at as much as 1000 megabits/sec. With 4/5 encoding and 40 bits transferred for every 32 bits of data, the deliverable bandwidth is 20 megabytes/sec. These links are used for data movement to/from the cluster from/to another cluster or an external source. Effective bandwidth is actually about 16 megabytes/sec after overhead is accounted for. The latency of a connection set up depends on when a message arrives with respect to a cycle boundary as explained in Section 3.

The Communication Interface consists of a Dual Port Memory (DP), a DMA controller, and an input/output FIFO buffer to input and output optical serial links. This subsystem is shared by all the PPs on a cluster. This avoids the cost of replicated communication hardware for each PP. Allocation of DP memory is controlled by the CC. The CC also controls the DMA controller through control registers, telling it what to transfer to/from the DP to the serial links. The DP can handle any combination of the following: reads from the serial link input FIFO buffer, writes to the serial link output FIFO buffer, and reads/writes from the shared memory bus. The FIFO buffers allow for speed matching between the DP accesses and the serial links data transfer rates.

2.3 The Group

At the next level of hierarchy, above clusters and PPs, are groups of clusters connected together though optical links and a crossbar. The clusters are also connected via the VME bus for control transfers. Separate groups can be connected with any topology as long as the required number of links per node including group I/O links does not exceed seven. That is because there are seven free links from the crossbar in a group. Each group consists of a Group Controller (GC), 8 Clusters, a spare Cluster for fault tolerance and a Generalized Communications Interface (GCI). The GC, a single board processor system equipped with a VME bus interface and an Ethernet interface, coordinates activities between the Group and the Group Host.

The Generalized Communication Interface: Connections on the crossbar are arbitrated by a Generalized Communication Interface (GCI) which receives all connection requests from the CCs and determines what to set up on a given communication cycle.

Controlling the GCI is an Intel i960. This processor was chosen to maintain compatibility between the Cluster and the GCI controllers. This made it easier to design and program the VME interface. The GCI controller's primary job is to arbitrate cluster communication requests and set up those connections on the Crossbar.

Connection request and grant information is passed to/from the GCI via mailbox registers on the VME interface chips. The GCI Controller then sets up those connections on the crossbar and grants transmissions once the connection is set.

The GCI controller also does the job of scheduling incoming images to clusters for processing. These incoming images may be placed in four frame buffers pending scheduling to the Clusters. These buffers are intended for data rate matching between incoming data sets waiting to be processed and the actual processing rate of the Group. Each buffer contains 256 kilobytes of DP memory and a connection to the crossbar. The frame buffer subsystems are configured exactly like the cluster communication interface except that they have less DP memory.

The crossbar consists of 20x20 connections. Nine of the connections are for use by the nine clusters. Four of the connections are for use by the frame buffers. The remaining seven are used for group interconnect and external sources.

The Group Controller: The Group Controller (GC) is a Sun Sparcstation VME controller card. It's main purpose is to coordinate actions over the VME bus and to provide the group with an interface to the Group Host through an ethernet connection. Operating system and all executable files are transferred by the GC from file systems to the clusters via the VME bus. All requests for group-to-group connections are passed through the GC from the GCI via the VME bus and are then forwarded to the Group Host via ethernet for arbitration.

3 The Proteus Communication Techniques

3.1 Proteus Communication Functions

Proteus uses a hybrid approach to communications, since the clusters are shared memory and the interconnections between clusters are message passing. We call the shared memory communication intra-cluster communications and the message passing communication inter-cluster

communications. Figure 2 shows the difference in the paths taken for these two type of communications. While the Cluster has a shared bus, an API to the programmer can give the illusion of a message passing system while retaining the speed of a shared bus. The result is a more uniform programming style.

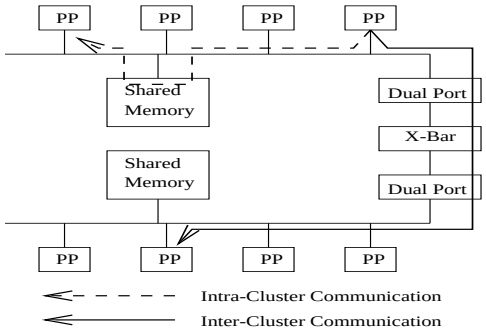

Figure 2: Proteus Communication Paths

Intra-Cluster Communications: Intra-cluster communication between PEs happens through shared memory. A simple way of looking at the intra-cluster communications is the classical cache coherency problem. The Proteus design maintains cache coherency through hardware-based, software-controlled, flush-and-sync mechanisms. Part of the PP subsystem design is a mechanism that when activated will selectively flush and/or invalidate certain locations in the cache. This mechanism is activated by a PP programming the control registers with the action to be taken (flush, invalidate, flush *and* invalidate).

The job of coherency is the responsibility of the programmer (or compiler). With larger grain tasks and messages, cache coherency maintained by software commands is a relatively low burden to the programmer (compiler) [15]. With more irregular fine grain applications this task can become increasingly difficult. However, Proteus is optimized for larger grain applications.

For an intra-cluster send, the sending PP just needs to flush the potentially cached data to an area of shared memory to make it available to another PP on the cluster. Once data is flushed by PP x it needs to let PP y know it is now available to it. By flushing the data and then setting a flag, PP x is free to continue with its computations. Care must be taken in making sure that the sending PP does not redefine the data at the location in shared memory before the receiving PP has received it. This would break multiple assignment rules. The data is now "owned" by the receiving processor and should not be modified by any other processor. A solution to this problem is buffering in which the PP makes its own copy of the data which it can modify as it pleases. Buffering needs to be used only if the same memory area is required by a processor for some other purpose. Alternate methods of consistency can be used but Proteus is programmed using rules based on a single owner per data block.

When PP y wants some data from another PP, it needs to check the flag to make sure the data has been placed there. If the checking of the flag is a blocking operation, then it is the same as a synchronization. This action is just a synchronization point. If the receiving PP checks in to see before the data has been sent, then it is blocked until the sending PP sets the check-in flag. If the sending PP has already checked in at the time the receiving PP checks in then the receiving PP can complete its receive and continue on with its computations. The receiving PP also needs to

know the address of where the data is. This address is passed back by the synchronization routine. The sending PP places this address in a predefined location as part of its check-in operation.

Inter-Cluster Communications: Inter-cluster communication is more involved than the intra-cluster, flush-and-sync communication. The communication is done on a synchronous, circuit-switched basis and occurs on a defined cycle basis. The CC acts as a controller for all interaction between a cluster and the crossbar network. A PP has to post a communication send request to the CC which then handles all control while the PP is free to continue with work.

The choice of synchronous circuit switching fits well with the original intent of the machine of macro-pipelining and permutation routing that is employed in volume visualization and other image processing algorithms. Research is being done to implement an asynchronous protocol on Proteus. The hardware is already in place and the protocol is being defined and refined.

The inter-cluster communication send mechanism is somewhat similar to the intra-cluster communication mechanism. In this case, the PP needs to place data in the DP memory area. Remember, the DP is directly addressable by the PP. This is much like the flushing part of the intra-cluster send. The network transmission part can be thought of as the synchronization between the sending cluster and the receiving cluster. Once this action is completed, the communication has occurred.

PPs communicate send requests to the CC through shared memory "mailboxes." When a PP wants to send data, it posts a send request in shared memory. This procedure fills a mailbox with communication information such as a destination processor, source address, and a communication ID tag. Once this action is complete, the PP is finished with the tasks it needs to do to communicate. Once again, to maintain multiple assignment rules, the data in DP should not be modified once it is assigned to be communicated. In fact, once a send is requested the sending PP releases ownership of that data block. This is very similar to the case for intra-cluster sends. Once the CC reads the request it handles all further action necessary to complete the send such as forwarding the request to the GCI and setting up the actual communication. This takes the burden of communication control off the PPs. Most of the time necessary to send data occurs during the period in which the CC is performing its actions. During this time the PP is free to do computations. This is exactly the concept of overlapping communication/computations.

For an inter-cluster receive, a PP *interrupts* the CC informing it that the PP needs data. An interrupt mechanism is used in order to allow the CC to respond almost immediately to the PP request. If a mailbox was used the CC could potentially have to wait until the next cycle to begin servicing the request. If the data was already in the DP, the delay is eliminated by the interrupt mechanism because the CC can immediately release the data to the PP. The data then sits in the destination DP until a receive for it has been issued. If a receive has been requested before the data has been received, the requesting PP has to wait until the receive occurs at which time the data is then released to the PP.

Uniform Communication API: To simplify programming, the API is fashioned so that communication calls that are made for local communications will invoke the flush-and-sync mechanisms without the programmer explicitly maintaining the flush-and-sync. The send or receive routines can be fashioned such that only one library function needs to be called for any communication (inter or intra) and the library function determines the kind of communication it is and takes the necessary actions. For ease of use, the determination of which type of communication, intra-cluster or inter-cluster, should be invisible to the programmer (compiler). In this

way coherency is maintained by message-passing coherency rules in the same fashion that inter-cluster coherency is maintained. The task of determining data movements has to be done for inter-cluster communications already so we maintain that having to determine these movements for intra-cluster communication adds no significant burden to the programmer (compiler).

The burden of determining what kind of communication is occurring (inter or intra) and maintaining that type of communication is taken off the programmer. In this way a single uniform communication API is presented to the programmer. This uniformity of communication commands is not always the most efficient method since an extra layer has to be added to intra-cluster communications to make it appear as message-passing in nature. Currently, this uniform API is not implemented. However, the structure is present in existing code to easily add this functionality to the library support.

3.2 Inter-Cluster Communication Specifics

While communication on a cluster is controlled between two PPs or by the CC in a uniform API case, communication between clusters within the Group is controlled by the GCI.

The movement of data among clusters is synchronized and each transmission is to be completed within a specified time which is defined before run time. When the term communication cycle is mentioned, it is only for inter-cluster communications since intra-cluster communications are asynchronous through shared memory and do not use a defined cycle. The control and data paths for inter-cluster communication are separate. Control is handled over the VME backplane while data is transferred via the crossbar during each cycle.

The Communication Cycle: To understand more about how communication occurs on a cycle basis, one needs to understand the actions taken during a cycle. At every synchronization cycle the CC reads transmission and reception grants from the VME register mailboxes that the GCI set up in the last cycle and sets up the communications accordingly. It then reads the requests that PPs have posted and forwards one of them to the GCI. Only one request is forwarded per cycle. The CC and GCI then reach a synchronization point and the CC is through with its preliminary work. The GCI control receives the connection requests from the cluster controllers during this cycle and arbitrates connection for the next cycle. The GCI then writes the VME register mailboxes with grant information for the next cycle. Any remaining requests that could not be serviced are queued by the GCI. The actions taken by the CC and GCI during a cycle are shown in Figure 3.

Four signals are used in defining a cycle. The baseline signal for a cycle is the End-Of-Transmission (EOT) signal. All other signals are defined in relation to this signal. Other signals include a signal to switch the crossbar (SW) and signals for the start of reception (RX) and the start of transmission (TX). Overhead is added by the synchronization time for the TX/RX circuitry for the optical links. The optical links we use have a rather large synchronization time resulting in a total overhead (latency) per cycle of 60 microseconds. With 64 kilobyte data packets, the effective data rate works out to about 15.98 megabyte/sec per link with a cycle time of 4.1 milliseconds. This represents a 71 percent efficiency of the peak data transfer rate of 20 megabyte/sec.

The 4.1 millisecond cycle time is necessary for those times that the communication resources are most busy. The exchange example in the next chapter is one of those times. In this case,

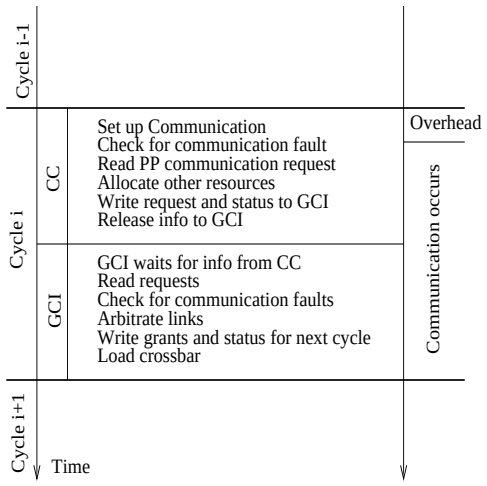

Figure 3: A Breakdown of the Proteus Communication Cycle

during any one cycle, a send and a receive are occurring and accesses are being made by more than 2 processors to the DP on the shared memory bus. This causes some contention for the DP resources resulting in a longer cycle time. It is suspected that a design error in the DP memory controller allowed this contention to occur. If designed correctly, it should take 3.125 milliseconds to transfer 64 kilobytes of data plus the 60 microseconds plus a small amount of overhead to complete the cycle. Sample tests show that the cycle time could be dropped to 3.325 milliseconds if the DMA had complete priority over the shared memory accesses to the DP. In these tests the DMA was set up to do a send and a receive on a cycle that the PEs were not accessing the DP from the shared memory bus. Cycle timing is shown in Figure 4.

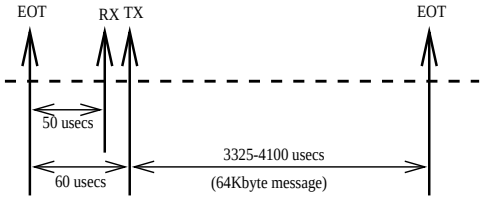

Figure 4: The Proteus Communication Cycle Timing

Request Arbitration: Priority among the PPs for forwarding requests to the GCI is determined by the CC on a round robin basis. For instance if the priority was (PP1, PP2, PP3, PP4) and PP1 had a send request, the next cycle would have priorities (PP2, PP3, PP4, PP1) for posting send requests. Requests are placed into the queue on a round robin approach similar to the CC-PP arbitration scheme. The GCI arbitrates from a request pool on a first-request/first-serve basis provided no conflicts exist. If a conflict exists for a given request, it stays in the queue and gets arbitrated on the next cycle.

Optimal Dual Port Performance: The method of involving the DP in the inter-cluster communications can impact efficiency in terms of shared memory access. This shared memory access can result in contention for the shared memory bus thereby degrading performance. The following code fragments show how to optimize DP access to minimize total memory accesses. "A", "B", and "C" are arrays of elements with size equal to the data packet and reside in shared memory. "DP" is the DP packet residing in DP memory. All operations occur for all elements in the arrays. Note that in the efficient case no unnecessary data movements are performed.

```
Efficient DP Use For Receive


     A = B + DP       {A <- B + DP}
     free_DP(DP)      {Free this DP area}


Inefficient DP Use For Receive


     C = DP           {C <- DP}
     free_DP(DP)      {Free this DP area}
     A = B + C        {A <- B + C}
```

It is easy to see the inefficient case resulting in extra copying. This concept of working to/from the DP area can be applied to the send routine to minimize data copies. Let us say a processor has to do *work_1* then send the results. If the final target area in DP is incorporated into *work_1*, then the data will not have to be copied to DP. These mechanisms are very similar to the receive mechanisms.

Communication Fault Tolerance: Fault tolerance is built into the communication system. Control signal errors and transmission errors are detected. An error in control signals leads to a mis-timed cycle. Transmission errors are detected by using a 4/5 bit data encoding scheme. Any link errors resulting in an erroneous receive packet are also detected. Any detected error is relayed to the GCI and a retransmission is scheduled.

3.3 Inter-Cluster Communications Flow Charts

Thus far, all the parts necessary to do an inter-cluster communication have been presented. The list of actions taken in a inter-cluster send are listed below. The flow chart for a send is shown in Figure 5.

1. A PP needs to send.
2. Send request is queued in shared memory mailbox registers. PP is free to continue on with computations.

3. The CC reads the request from a PPs' queue if it has top priority . If another PP has a higher priority it's request is serviced and the other request stays queued.

4. CC posts the request to the GCI.

5. GCI reads the request from the CC mailbox registers.

6. GCI places request in an arbitration queue.

7. If a grant is determined, the send request is set up. If not, the request stays in the arbitration queue.

8. Send occurs.

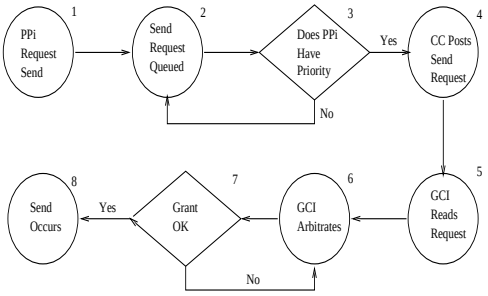

Figure 5: Send Flowchart

The minimum latency occurs when 1, 2, 3, 4, 5, 6, and 7 happen on the same cycle and 8 happens on the very next cycle. Higher latency occurs when the request does not have priority and gets queued at steps 3 and 7. These two steps increase the latency by a multiple of the cycle time.

The list of actions taken in a inter-cluster receive are listed below and the flow chart for a receive is shown in Figure 6.

1. PP issues a receive.

2. Receive request gets queued.

3. If the corresponding send has occurred then resolve it and release data to PP. If not, the request stays in the queue.

4. Receive occurs.

The minimum latency occurs when 1, 2, 3, and 4 happen on the same cycle, that is, the corresponding send has already occurred. Higher latency occurs when the send has not occurred. The latency will then increase by some multiple of the cycle time.

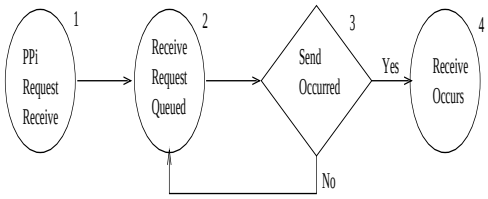

Figure 6: Receive Flowchart

4 Overlapping Communication/Computation Techniques

4.1 Ideal Communication/Computation Overlap

In order to overlap communications with computations, the PPs would be able to do work while sends are occurring and before receives are completed. Sends should be initiated as soon as possible. The time after a send should be filled in with tasks independent of the sent data to maintain data consistency. Receives for the PP, should be posted as late as possible, with tasks independent of the receive scheduled before it. By scheduling a receive as late as possible, the actual data can take place without affecting the task as long as possible, thus keeping the receiving processor from waiting too long. This order of events is shown in figure 7. If enough work can be filled in before a receive and after a send so as to keep the processors fully utilized, then optimal overlapping communication/computation has been achieved.

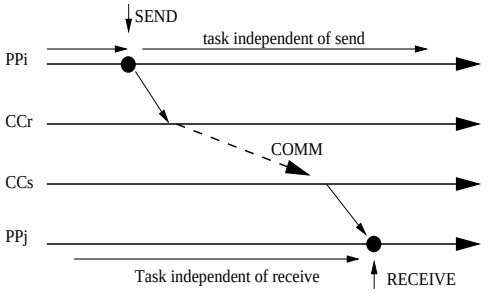

Figure 7: Proteus Communication Scheduling

Overlapping communication/computation addresses the primary goal of keeping processor utilization high at all times. One way of looking at overlapping communication/computation is the following way. A block of work is defined as a task. For a given program a processor has many such tasks. Each task is assumed to have some communication requirements associated with it. A task that has no communication associated with it has communication requirements with zero time. For a task that has multiple communications associated with it the sum of the time is the communication requirements for that task. Ideal communication/computation overlap occurs in the following situation. While working on a given task i, the incoming communications for task $i+1$ are occurring. The outgoing communications from task $i-1$ can also be scheduled to occur during this time. If the sum of the incoming communications for task $i+1$ and the outgoing communications for $i-1$ is less than the computation time for task i, then the communication time is effectively hidden. This scenario is depicted in Figure 8(b). Figure 8(a) shows a diagram

of no communication/computation overlap. This would be a case where a processor had to serve all of it's communication and could only do one thing at a time. This is also the case when a processor cannot fill in the time during a send or receive with computations. If the communication time is more than the computation time during task i, then incomplete overlap occurs. Figure 8(c) shows incomplete communication/computation overlap in which there is more communication for a given task than computation. This problem is said to be communication bound.

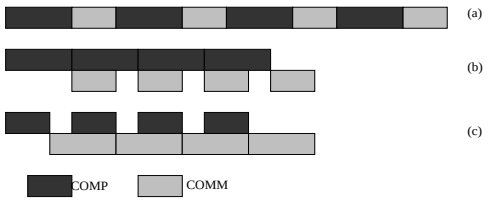

Figure 8: Communication scheduling style: a) serial communication/computation, b) complete overlap of communication by computation, c) incomplete overlap or communication bound.

The goal is to partition the program in such a way that the task time can hide the communication time. A related goal is to schedule the communication efficiently in a manner that overlapping communication/computation can occur. This chapter shows even the simplest tasks can result in an efficient overlapping.

4.2 The Exchange Example

The result of the CC handling the communication functions is that the PP is free to concentrate on computation tasks. For instance, a PP can post a send request, then continue to work while the CC handles the chores of setting up the transaction. However, for a cluster on Proteus, a single PP is unable to fully utilize the communication links and the CC resources by itself. Hence a cluster of four processors is used to utilize the capacity in full.

Communication Resource Utilization: The best way to demonstrate an efficient order of events and efficient communication resource utilization of Proteus is through an Exchange example. In this example, a single PPi on one cluster exchanges a 64 kilobyte data packet with a single PPj on another cluster. At any given time, two data packets are bouncing back and forth between the two PPs. When the packet arrives, some work (a simple check-and-modify operations) is performed, and the packet is sent back to the other PP. Figure 9 depicts what happens in the exchange example. In cycle q, both CCi and CCj set up to send and receive a packet. In cycle q+1, the packets have been received and are released to the PPs. The PPs work on the packet through the cycle q+2 and posts a send. In cycle q+3, the CCs read the PPs posted send and requests a connection to the GCI. The GCI arbitrates and schedules a transmission in cycle q+4. The cycle then repeats with the other packets similarly.

Altering the test to include 4 PPs on each cluster and exchanging data packets with 4 PPs on another cluster, results in an order of events similar to the single PP case except after an initial clash for resources the PPs will step into order and utilize the cycles not used by other PPs. The link is utilized 100% of the time with a transmit and receive occurring in every cycle.

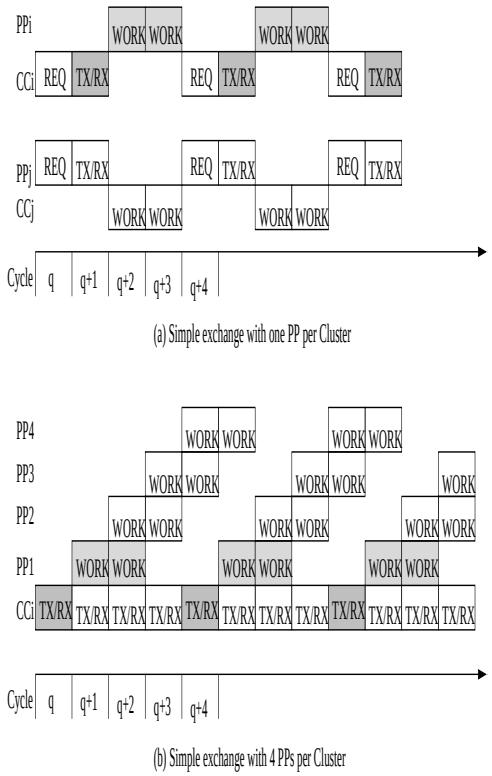

Figure 9: Exchange Example a) one PP per cluster b) 4 pps per cluster (shading is work of one PP

This scenario is shown in Figure 9. In intensive applications, resources not used by one PP are free to be used by the other PPs. Therefore, a major goal of efficiently sharing resources such as communication hardware and memory has been met without adding overhead due to contention. So far only the utilization of the communication resources of Proteus has been highlighted. Real life programs have the computation being more involved, so the example is restructured to take advantage of overlapping communication/computation features.

Implementing the Overlapping Communication/Computation: A more realistic example would entail the work portion taking many more cycles than the simple check and modify operation. An assumption is made that real life calculation would be at least twice that of the simple check-and-modify, or about 4 cycles as shown in Figure 10(a) for a single PP case. This modified exchange example still has the problem of the PPs laying idle waiting for the next set of data.

A change is now made in the scheduling of communications so that the PP transmits its data after finishing half of the data packet of size B. This modified schedule of events is shown in Figure 10(b). Again a single PP is shown. As in the simple exchange model, when scaled to four, the PPs will step into place and now both PPs and the CCs are fully utilized. This is shown in 10(c).

By splitting the data and sending B/2 sized packets, the PP is now able to maintain full efficiency. From the previous examples we see that for the Proteus machine, with its 4 PPs per cluster, a task needs to be at least 4 cycles long for each communication (send and receive) scheduled in order that the PP maintains full computation utilization. If the task were any shorter the communication links would be fully utilized while the PPs would lay idle waiting for data. In this case, we have a communications-bounded problem. If the task were any longer, we still get 100% computation utilization with the communication time hidden. We see in the next section that this is a very realistic goal.

Dividing the working data size in half is effective only up to a certain extent. Each division causes the creation of another packet to be sent which has more overhead associated with it. Once complete overlap is achieved, additional splitting results in no benefit. In fact, oversplitting can cause many more data packets which can overload the communication system resources.

Conclusions from the Exchange Example: In the exchange example the efficient utilization of all communication resources is demonstrated. As the number of PPs in the exchange was increased, the communication resource utilization was maintained and no bottleneck was created. Through the use of splitting techniques, the utilization of the PP system was increased while still avoiding a bottleneck in the communication system. The techniques of partitioning tasks and scheduling communications learned in this section can be applied to real applications. In the next section, we demonstrate the use of these techniques in a Two Dimensional FFT Algorithm.

4.3 The FFT Example

The application we use to evaluate the impact of overlapping communication/computation is the 2-Dimensional FFT. The algorithm we use, based upon an algorithm found in [9], can be split into five main parts:

1. The second dimension bit reversal - line swaps.

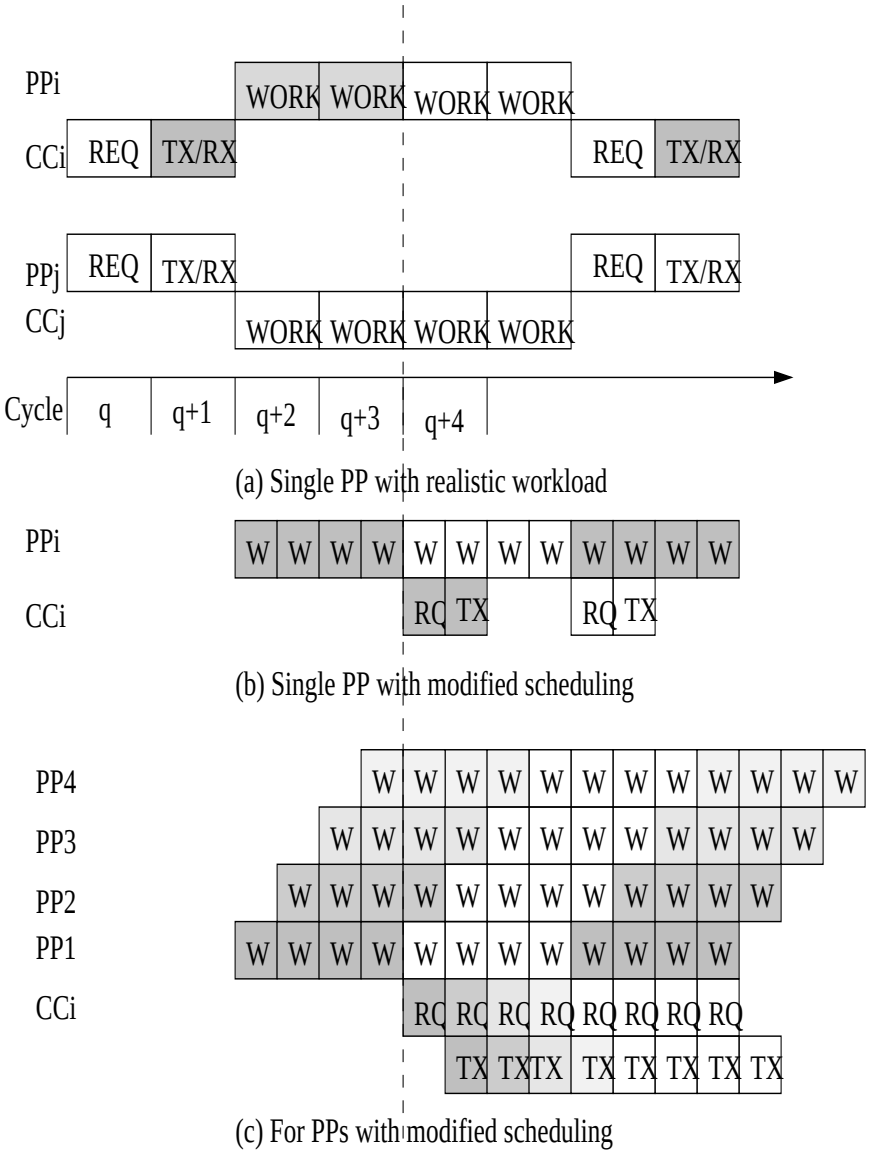

Figure 10: Exchange example with more realistic workload a) single PP, b) single PPs with modified scheduling, (c) four PPs with modified scheduling

2. The distribution of data.

3. The first dimension FFT and initial intra-grain combines for second dimension.

4. The inter-grain second dimension FFT.

5. The collection of the data.

We implemented this algorithm on the Proteus machine. We used the 2 dimensional FFT "C" code in [10] as a starting point and parallelized it while maintaining the spirit of the code. The greatest emphasis was on optimizing the communication patterns. On Proteus a "master" node was used to perform the bit-reversal and distribution. Then n nodes are used to do the FFT in both dimensions. Finally, each node sends its processed data back to the master node where it is reconstructed into continuous memory space. Our goal was to overlap as much computation and communication as possible.

In order to achieve overlapping we decided to partition the data into coarse grains, setting up a pipeline of sorts with computation and communication. Parts 1 and 2 of our algorithm can be overlapped in the following way. Bit reversed grains of data are formulated from the complete data set. After one is formulated, it can be sent off to a processor to begin the FFT computations. While it is being sent off, the next grain can be formulated. We see from the exchange example that the read and write operation involved in formulating a bit-reversed grain can indeed be overlapped with communication on Proteus. In fact since the master processor is just reading, writing, and sending we observe that the communication channel is utilized 100% with grains being sent to various processors every cycle.

Parts 2 and 3 of our algorithm can be overlapped in the following way. As each processor receives the data, it is able to process the first dimension FFT and the first few butterfly stages of the second dimension FFT before and while it is receiving its next grain. When the processor finishes it's computation on the grain, the next grain is already there on which to begin processing.

Parts 3 and 4 of our algorithm can be overlapped in the following way. As grains are finished with their initial dimension 2 processing (butterflies within the grain) they can be sent off to their destination processors. When the processors finish the initial dimension 2 processing, the incoming grains that need to be combined with other grains should already have arrived.

Overlapping within part 4 occurs in the following way. Once a processor receives all of its grains and processes as many second dimension FFT butterfly stages as it can, a processor then needs to perform the remaining stages of the butterfly combine operation with other processors. This means each processor needs to send and receive data from other processors. Here we can take advantage of the overlapping computation/communication features of the two target machines. A processor can do the combination of a received grain while receiving the next grain for the next computation. In this fashion the processors complete the remaining stages of the 2D FFT.

Parts 4 and 5 can be overlapped in the following way. When completed, the data is collected by the master processor and reconstructed into continuous memory space. As the grains are completed with all the processing, they can be sent back to the master processor for reconstruction. This can occur while other grains are being processed. This stage is much like the first stage except the data is moving the other way.

Timing Example from the 2D FFT: The previous section shows where savings can be made to reduce communication overhead. In this section we analyze one of those cases to further demonstrate the effects of communication/computation overlapping. This analysis is of the overlapping

that occurs within part 4 of our algorithm. The routing permutations in the second dimension FFT were chosen to reduce communications. Even with overlapping communication and computation, the communication time could have an effect on the overall time if the communication time was much bigger than the computation time. So the goal is to reduce communication as much as possible and also overlap the remaining communication with computation. Upon first examining the communication patterns of a butterfly stage for two processors with two grains each, we see that the two processors could send each other their grains, then do the combination to produce two grains each. See Figure 11(a) for details. Upon closer examination, if a processor has two grains, it is seen that the first processor could send its upper grain to the second processor and the second processor could send its lower grain to the first processor. Each processor could then combine the incoming grain with the grain it kept to produce two grains each. See Figure 11(b) for details. This results in less overall communication between processors. A byproduct of this rearrangement is that the multiplies that occur as part of the FFT computation are split amongst the processor involved resulting in a load balanced approach. Furthermore, each processor could split its data to utilize overlapping communication and computation as shown in Figure 11(c). This results in more overall communication between processors but the data size communicated per transaction is halved and the communication is effectively hidden.

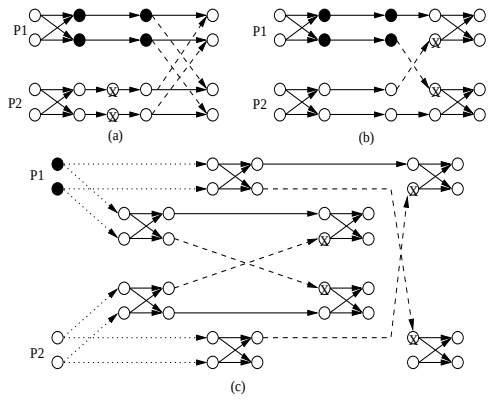

Figure 11: a) Tradition butterflies, b) Optimized butterflies, c) Optimized butterflies with overlapping communication/computation.

This technique reduced the second dimension FFT communication in half. Another result of this reduced communication is a load balancing. In the FFT, "upper" grains do not need to be multiplied by a phase factor, while "lower" grains do. In the non-reduced computation case the upper node would do half as much work as the lower node. With the reduced computation each node will do both an upper combination without phase factor multiplication and a lower combination with phase factor multiplication. In a 32 processor scenario on Proteus with 64 kilobyte grains we see that the processing (combining one resident grain with one incoming grain to produce 2 grains) time to be about 16 ms with grains able to come to a processor every 16.4 ms (4.1ms per grain with the other 3 processors on board also needing to receive a grain = 4 processors * 4.1 milliseconds = 16.4 ms). In the non-reduced communication scheme the processing (2 grains producing 1 grain) would take 8 ms with grains coming every 16.4ms. We see in this implementation, a processor would spend much of the time waiting for incoming grains since the communication time dominates. These numbers support the design decision to supply

four processors on a cluster board. Any more processors would result in the communication needs not being met. Any fewer and the communication resources would lay idle much of the time.

Bandwidth Requirements: In a paper by Sivasubramaniam et al. [11], the authors discuss bandwidth requirements for various parallel applications. Using the SPASM simulator, they were able to characterize the bandwidth necessary to reach certain levels of overhead associated with communication time. They also used processor clock rate as a function to their bandwidth requirements. They computed bandwidth requirements for five applications for clock speeds of 33, 100, and 300 Mhz for levels of overhead of 50%, 30%, and 10%. However, this paper fails to account for overlapping communication/computation possibilities.

The authors suggest that to meet a 10% restriction on overhead (90% efficiency), the bandwidth requirements for an FFT problem would be characterized by $0.01p^{0.36} + 16.37$ megabytes per second for a 33 Mhz processor. This corresponds to an effective bandwidth of 16.87 for a 32 processor system. A 30% restriction in communication overhead is presented as 7.83 megabytes per second in this work. For this implementation to approach 0% communication overhead, requires a substantial increase in effective bandwidth. This particular implementation of the FFT uses no overlapping communication/computation and considers the core processing and not the distribution time. This 16.87 is used as a baseline bandwidth figure in a comparison with the Proteus implementation though the bandwidth requirements would be raised slightly for the 40 MHz i860. Examining the 2D FFT implementation on Proteus, and only considering the core FFT computations, the 2D interprocessor butterfly stages are the only communication that take place. Proteus already has 16 megabyte per second communication links. By overlapping the communication and computation we are able to completely hide the communications. So the communication overhead in the Proteus implementation is able to approach 0% without the need for more communication bandwidth.

Another algorithm presented in [11], CHOLESKY, needs a bandwidth increase from 16.02 to 84.12 in order to reduce communication overhead from 30% to 10% for a 32 processor system with a 33 MHz node. The IS (Integer Sort) example with the same system configuration needs an increase in bandwidth 78.61 to 211.45. So in reducing communication overhead in this approach, the bandwidth requirements need to be increased greatly. Such drastic increases in bandwidth requirements needed to decrease the effects of overhead may be unnecessary if the use of overlapping communication/computation is explored for these applications.

Conclusions from the Two Dimensional FFT: From the results that we have presented, we see the tradeoff between the reduction in communication overhead due to large granularity and the effects of overlapping computation/communication. By using the concept of overlapping computation/communication we are able to save on the overall time of the algorithm. Instead of a processor having to do communication then computation, we reduce the overall time by ensuring that the time a processor spends communicating (overhead) is minimized through overlapping.

5 Complete Exchange in the Proteus System

In a complete exchange among k processors, numbered 0 to $k-1$, each processor i has a unique message for every other processor j, $0 \leq i,\ j < k$. There are three main algorithms used to

achieve complete exchange among processors in hypercube- and mesh-like machines: standard, direct, and multi-phase.

In standard exchange algorithm for a hypercube [18], data meant for multiple destinations are sent to a neighboring node by each node. The receiving nodes shuffle the data around and send them to their neighbors in another dimension ensuring that the data move towards their destinations. In a hypercube, it takes $\log k$ transmissions of messages of size $m * k/2$ each where m is the size of data to be sent from one node to each of the other nodes. Each node has to shuffle the blocks of data of size $m * k$. In the direct algorithm, each node sends $k-1$ individual messages of size m to $k-1$ other nodes [19] using a straightforward algorithm. It turns out that for large messages, direct algorithm outperforms the standard exchange. This is because manipulation of messages on the intermediate nodes is more expensive than letting the messages go untouched. In the multi-phase algorithm [20], a complete exchange is obtained by doing two or more partial exchanges. Each partial exchange is a complete exchange in a subcube. Similar algorithms have been implemented on other machines such as SP2, Paragon, and CS-2 [20] and have been shown to be very effective.

The time to finish a complete exchange on the three machines, Paragon, SP-2, and CS-2, using multi-phase algorithms are reported in [20] and are as shown in Table 1. In the table, we have picked up the best times possible. It should be noted that some of these times do not scale well depending on the multi-phase strategy. In some cases, the strategy may not be able to handle larger message affecting the times adversely. Experiments on SP2 and CS-2 up to 64 processors. It should be noted that individual message times do not scale well when multiple messages are transmitted at the same time due to contention in the network.

Table 1: Complete Exchange Times on Paragon, SP-2, and CS-2 [20]

No. of Proc	Message Size	Paragon	SP2	CS-2
Single Message time	m bytes	75+0.011 usec	70+0.043 usec	105+0.025 usec
32 Proc	2K bytes	7 milliseconds	7 milliseconds	11 milliseconds
64 Proc	2K bytes	12 milliseconds	13 milliseconds	28 milliseconds
128 Proc	2K bytes	40 milliseconds	–	–
256 Proc	2K bytes	88 milliseconds	–	–

Since each cluster in Proteus system has four processors and each group has eight clusters, a 32 processor configuration is a single node with full crossbar connection between the eight clusters. To derive any larger configuration, we have several choices. One can start with as many full groups and a partial group and then interconnect them using the seven links available at each group for interconnection. A few example structures are shown in Figure 12 below.

We assume that no two groups are connected using more than two links (there is no restriction as long as no more than seven links are required to connect a group in the structure). In the following we will also arbitrarily assume that the interconnection structures are hypercube-like which was our original intention at the time of design.

Recall that each cluster can transmit and receive at most one message in each time slot. Thus

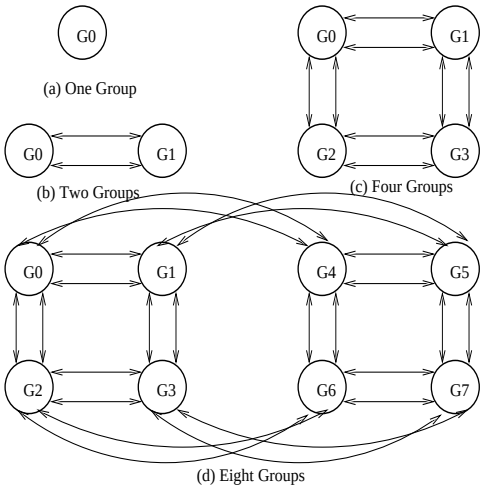

Figure 12: Some configurations of the Proteus system

eight clusters in each group can potentially transmit and receive up to eight messages in each time slot. However, only a few of them can be sent out at a time as the number of external links to and from each group are limited to seven and all of them may not be connected. The bisection bandwidth of the structure used will govern the amount of time it takes to achieve a complete exchange.

For a complete exchange on the Proteus, we use a strategy where messages from the four processors on a cluster are accumulated as a single message. This can be easily done, since the processors share memory. Thus, each cluster will have a unique message for every other cluster in the system. These messages are then transmitted to the destination clusters. The process is repeated overlapping communication and assembly of messages. Thus each cluster prepares one message, receives and un-assembles one message, and transmits one message in most of the cycles. Since each group can only transmit only a limited number of messages in a cycle, there will be some idle cycles on all clusters during the exchange.

Number of Messages. Each cluster in a group has some inter-group messages and some intra-group messages. Intra-group messages can be transmitted in a conflict free manner, as long as it is a permutation. Inter-group messages need to be scheduled. If a machine configuration has k groups ($8 * k$ clusters), then each cluster needs to send $8 * (k - 1)$ inter-group message and 7 intra-group messages. Since there are 8 clusters in a group, the number of inter-group messages from each group is $8 * 8 * (k - 1)$. The number of intra-group messages from all clusters together within each group is $8 * 7$. Moreover, a group can at most transmit as many inter-group messages as the number of outgoing links or the number of clusters in the group. Since in most configuration the first one will be the case, we find a schedule that can keep outgoing links busy as much as possible.

Schedule. If $k = 1$, then it will take exactly 7 cycles to finish communication within a group. For $k > 1$, it is the inter-group traffic requirement that will govern the total time. The intra-group communication can be completely overlapped with the inter-group communication for $k > 1$. The structure in Figure 12 is a generalized folding cube. For a 8 group configuration as shown in

Figure 12(d), the following schedule in Table 2 will be able to keep all nodes busy in all cycles in a regular hypercube. The routing is based on the *e-cube* routing algorithm [21]. In general a k node configuration will take $k-1$ cycles if each node transmits exactly one message in each cycle. This is the direct algorithm for a complete exchange.

Table 2: Permutation Routing Schedule on A Hypercube

Number	Mask	From $\rightarrow$ To Pairs
0	7	(0, 7), (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1), (7, 0)
1	6	(0, 6), (1, 7), (2, 4), (3, 5), (4, 2), (5, 3), (6, 0), (7, 1)
2	5	(0, 5), (1, 4), (2, 7), (3, 6), (4, 1), (5, 0), (6, 3), (7, 2)
3	4	(0, 4), (1, 5), (2, 6), (3, 7), (4, 0), (5, 1), (6, 2), (7, 3)
4	3	(0, 3), (1, 2), (2, 1), (3, 0), (4, 7), (5, 6), (6, 5), (7, 4)
5	2	(0, 2), (1, 3), (2, 0), (3, 1), (4, 6), (5, 7), (6, 4), (7, 5)
6	1	(0, 1), (1, 0), (2, 3), (3, 2), (4, 5), (5, 4), (6, 7), (7, 6)
7	0	(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7)

Optimization for the Proteus System. In most implementations of hypercube-based architectures each node transmits only one message at a time, *e-cube* routing works very well and keeps maximum number of links busy during the realization of a permutation. A close look at these permutations shows that links used by permutation pairs 6 and 1, 5 and 2, and 4 and 3, are mutually exclusive. Thus if it was possible to transmit more than one message from a node, we could merge these pairs of permutations to realize a better utilization of links. In the Proteus system, we can transmit more than one messages from each group (but from different clusters within the group), we can embed a pair of permutations while still using *e-cube* routing. Thus, it takes only $k/2$ cycles to realize all the permutations. In fact, while performing permutation 0 in Table 2, the clusters could also perform permutation 7 with mask 0 realizing an intra-cluster transfer. A similar approach will work for lower and higher dimension structures.

Thus far we have used only a single bidirectional link in the hypercube structure of the Proteus. If we have l links in each dimension(as shown in Figure 12 for $l = 2$), we can further combine the sets of permutations being realized in two cycles in one cycle as they will use parallel links and there are enough sources (clusters) available to transmit messages within a group. In the case of the Proteus, l must not exceed 4. This will reduce the number of cycles from $k/2$ to $k/2l$ for one unique message from one group to every other group.

Embedding Complete Exchange. We need to transmit 8 messages from each cluster to 8 clusters in the other group and there are 8 clusters to transmit in each group. Therefore, we need to transmit 64 unique messages from each group to every other group. The number of cycles it will take to achieve a complete exchange in a k groups system with l links is given by $64 * k/(2 * l)$ for $k > 1$. For $k = 1$, there is no inter-group messages and intra-group permutations can be

achieved in 7 cycles. In general if there are c clusters per group, then the time taken is given by $c^2 * k/(2 * l)$ for k groups and l links in each dimension system for $c > 2l$. For $k = 1$, it takes $c - 1$ cycles.

Results. Now we compute and compare the time it takes to achieve a complete exchange on various machines. As reported earlier, in Proteus, to transmit a 64K message, theoretically it should take 3.2 milliseconds + 0.1 milliseconds to set up the connection for each permutation cycle. However, due to hardware speed mismatches, it takes 4.0 milliseconds + 0.1 milliseconds (a total of 4.1 milliseconds). For a message size of 32 K this time will be 2.0 milliseconds + 0.1 milliseconds = 2.1 milliseconds as it is all governed by the hardware speed once the message is in the network. Recall that there is no contention in the network. Similar computation gives a total time of 1.1 milliseconds for 16 K and 0.6 milliseconds for a 8 K message size. If an individual processor generates a message of size m bytes, then a cluster will generate a message of $4m$ bytes. For $m = 2$K from each processor to every other processor, we have 8K byte messages from each cluster. The actual time it takes with overlapping communication and computation is given in Table 3. Notice that unlike standard exchange or multi-phase exchanges, the messages here can be computed and prepared as the algorithm progresses as is the case for the direct algorithm. The times in Table 1 must be compared with the times in column corresponding to the message size 2K in Table 3.

Table 3: Complete Exchange Times (in milliseconds) on Proteus, $c = 8$

Proc	k	$l = 1$				$l = 2$			
		2K	4K	8K	16K	2K	4K	8K	16K
32	1	4.2	7.7	14.7	28.7	4.2	7.7	14.7	28.7
64	2	38.4	70.4	134.4	262.4	19.2	35.2	67.2	131.2
128	4	76.8	140.8	268.8	524.8	38.4	70.4	134.2	262.4
256	8	153.6	281.6	537.6	1049.6	76.8	140.8	268.8	524.8

Several things are noteworthy here. First, the message sizes (from each node) are scalable. This is generally not true for most machines which are designed with small messages in mind. Second, the time it takes to deliver all messages is also scalable. For $k > 1$, the time simply multiplies by a factor of two when k is multiplied by a factor of two. This is not the case with the times measured on Paragon, SP2, and CS-2. Third, for larger configurations, Proteus takes a much shorter time than the Paragon, SP2, or CS-2. It should also be noticed that the link speeds are the smallest on the Proteus machine among all the machines compared here. Thus, it is not the raw speed, but the organization that matters. Also, recall that the algorithms here will scale for $l \leq c/2$.

Further Optimization. We could use the multi-phase approach of Bokhari to further optimize the complete exchange on the Proteus. In this case, first we accumulate all messages within a group that are to be sent to other groups on different clusters and then transmit the messages. This

allows us to transfer fewer messages of larger granularity. Thus the set up time can be minimized. However, one should be careful with the overhead in collecting messages. Our suspicion is that, like in direct vs standard exchange algorithm, message overhead may wash out the set up time gain. We, therefore, do not recommend it.

6 Conclusions

This paper has explored the use of overlapping communications/computations and their implementation on the Proteus parallel computer. Specifically, a detailed description of the hardware that Proteus provides to accomplish overlapping communication/computation has been presented in section 2. Dedicated hardware in the form of a communications controller takes the burden of communications control away from the working processors, thus freeing the processor to concentrate on computations. On Proteus, this hardware is shared by the four processors in a cluster, thereby reducing overall costs. Efficient utilization of this communication processor is maintained without overloading it with communication functions. We outlined the software work done on Proteus to support overlapping communication/computations in Section 3. Specifically, we addressed the issues of library support and required actions during a communication cycle.

Ideally, communication and computation should be scheduled in such a way that the computation time completely overlaps the communication time. The exchange example demonstrates the ease in scheduling overlapping communications/computations while keeping communication utilization from saturating. The processors do not have to idle while waiting for communications to occur since they are continuously kept busy. This example shows a method of splitting a task to break the serial communication/computation pattern. This example also shows that even for a relatively simple task coupled with many communications, overlapping communications/computations are able to maintain efficient processor and communication subsystem utilization. The results of this example strengthen the choice of sharing the communication subsystem among four processors in Proteus.

A major issue addressed in this paper is that overlapping communication/computation is an efficient method to reduce communication overhead. Other attempts call for communication connections with higher effective bandwidth. The major drawback in these attempts is that the reducing overhead is not a linear function with respect to bandwidth increase. A rather large increase in an effective bandwidth is necessary to decrease the overhead to an acceptable level. Overlapping communication/computation allows for a reduction in the overhead without unacceptably large increases in the effective bandwidth. Using the two dimensional FFT example, a program is dissected showing overlapping techniques in use. This splitting method is applied in order to achieve complete overlapping of the communication and computation. For the second dimension butterflies that require communication, the communication overhead approaches zero without the need for additional bandwidth. Also, a reduction in overall execution time is achieved.

We also addressed the issue of achieving a complete exchange in the Proteus machine. We have shown that even with much smaller bandwidth per processor (approximately 4Mbyte/sec per PP or 16Mbyte/sec per cluster in contrast to 175Mbyte/sec in the Paragon) links, the Proteus machine can achieve a complete exchange in compararble or shorter time depending on the message size. Moreover these times are scalable with the machine size.

References

[1] R. C. Agarwal, F. G. Gustavoson, and M. Zubair, "An Efficient Parallel Algorithm for 3-D FFT NAS Parallel Benchmark," in Proc. of Scalable High-Performance Computing Conference, Knoxville, TN, USA, 1994, pp. 129-133.

[2] U. Brueing, W. Giloi, and W. Schroeder-Preikschat, "Latency hiding in Message-passing Architectures," in the Proceedings of the Eighth International Parallel Processing Symposium, Cancun, Mexico. pp. 704-9, April 1994.

[3] S. B. Choi and A. K. Somani, "Rearrangeable Circuit-Switched Hypercube Architecture for Routing Permutations," Journal of Parallel and Distributed Computing, Vol. 19, 1993, pp. 125-133.

[4] W. Groscup, "The Intel Paragon XP/S supercomputer," in the Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology, Parallel Supercomputing in Atmospheric Science, Reading, UK, Nov. 1992, pp. 173-87.

[5] M. Harrington and A. K. Somani, "Synchronizing Hypercube Networks in the Presence of Faults," IEEE Transactions on Computers, Oct. 1994.

[6] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," in the Proceedings of ASPLOS VI, San Jose, CA, pp. 38-50, Oct. 1994.

[7] J. Kuskin et al.,"The Stanford FLASH Multiprocessor," in International Symposium on Computer Architecture, pp. 302-313, 1194.

[8] D. Lenoski et al.,"The Stanford DASH Multiprocessor," Computer, Vol. 25, pp. 63-79, Mar. 1992.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical Recipes in C, Second Edition," Cambridge University Press, New York, NY, 1992.

[10] A. Sansano and A. K. Somani, "The Communication System of the Proteus Parallel Processor," in the Proceedings of the International Conference on High Performance Computing, New Delhi, India, 1995, pp. 635-640.

[11] A. Sivasubrmaniam et al., "On Characterizing Bandwidth Requirements of Parallel Algorithms," in the Proceedings of Sigmetrics 95/Performance 95, Ottawa, Canada, pp. 198-207, 1996.

[12] A. K. Somani, et al., "Proteus system architecture and organization," in Proceedings of the Fifth International Parallel Processing Symp., Anaheim, CA, April 30 - May 2, 1991, pp. 276-284.

[13] H. S. Stone, "High Performance Computer Architecture," Reading, MA, Addison Wesley, 1987.

[14] T. Tarui et al., "Evaluation of the Cluster Structure on the PIM/C Parallel Inference Machine," in the Proceedings of the 1994 International Conference on Parallel Processing, Vol. II, pp. 309-313, 1994.

[15] J. Torrellas and D. Koufaty, "Comparing the Performance of the DASH and Cedar Multiprocessors for Scientific Applications," in the Proceedings of the 1994 International Conference on Parallel Processing, Vol II, pp. 304-308, 1994.

[16] C. M. Wittenbrink, A. K. Somani, and C. -H. Chen, "Cache write generate for parallel image processing on shared memory architectures," in IEEE Transactions on Image Processing, Vol. 5, No. 7, July 1996, pp. 1204-1208.

[17] i860 64-Bit Microprocessor Hardware Reference Manual, Mt. Prospect, IL, Intel Corp., 1990.

[18] S. L. Johnson and C. -T. Ho, "Matrix Transposition on Boolean n-Cube Configured Ensemble Architectures," SIAM J. Matrix Analysis Application, Vol. 9, No. 3, July 1988, pp. 419-454.

[19] R. Take, "A Routing Method for the All-to-All Burst on Hypercube Network," Proc. 35th National Conf. Info. Proc. Soc., Japan, 1987, pp. 151-152 (in Japanese).

[20] S. H. Bokhari, "Multiphase Complete Exchange on Paragon, SP2, and CS2," in IEEE Parallel and Distributed Technology, Fall 1996, pp. 45-59.

[21] S. L. Johnson and C. -T. Ho, "Optimum broadcasting and personalized communication in hypercubes," IEEE Trans. Comput., Vol. 38, Sept. 1989, pp. 1249-1268.