User-Level Communication in a System with Gang Scheduling

Yoav Etsion and Dror G. Feitelson

School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{etsman,feit}@cs.huji.ac.il

Abstract

One of the main limitations on multiprogramming of parallel jobs on clusters is the communication bandwidth allocation. When using an off-the-shelf network card there are memory limitations: the memory size resident on the network card is fixed. Because of this, any use of this memory as a fast send or receive buffer prohibits us from using a multiprogramming schene because of the obvious need to divide the buffer among the different jobs, which reduces the achievable bandwidth substantially. In this paper we propose a different scheme, which combines gang scheduling (coordinated context switching of the processes on different nodes) with a communication buffers switch. In this scheme, we associate a communication buffer with each context. Whenever the scheduling algorithm initiates a context switch, we also replace the communication buffers — the send queue and receive queue. This allows the jobs to utilize the bufers as if it was the only job in the system, resulting in improved bandwidth.

1 Introduction

It is widely known that buffer sizes affect communication performance. The buffer size dictates the size of both the receive queue and the send queue. These sizes correlate with the flow control window size — the number of messages that can be sent before an acknowledge reply is required. Most of today's flow control algorithms use such windowing schemes, including e.g. TCP.

Even though the communication buffers sizes are important and small buffers limit the maximum achievable bandwidth, they cannot be as big as we may want them to be because there are several limitations on the buffers' size:

- The computer has limited memory resources. If too much of the memory is allocated for communication buffers the application itself will be short on memory, resulting in paging overhead that will eliminate the gain from the higher bandwidth.

- Most implementations and drivers use some of the buffers as DMA buffers. Such buffers must be pinned down in memory and cannot be paged. Also, these buffers must reside in a fixed physical location since the DMA engine accesses it using physical addresses. Too large buffers will limit the OS memory management, so again the application will incur a very large paging overhead.

- It is very common for implementations to locate some of the buffers on the network card itself. These cards have very limited memory resources — up to a few megabytes. This limitation is strict, as opposed to the previous ones which can be tuned to find an optima.

When designing a multiprogrammed parallel system the communication buffers pose a major dilemma. Each node in the system will run several processes that are parts of different parallel applications, each of which must have it's own communication context (so every application can act as if it was the only one on the machine). However, each node has a limited size of memory that can be used as communication buffers, and that must be used by all processes on the node. The most trivial solution that is commonly used is to divide the buffers by the number of processes, so each process will get exclusive use of its fair share of the buffers. Let us assume that the size of the send and receive buffers are B_s and B_r, respectively. If the number of processors is p and the number of processes running per processor is n, the effective send buffer size for each process is $\frac{B_s}{n}$ (the general send buffer is simply divided equally and exclusively among the running processes), and the effective receive buffer will be of size $\frac{B_r}{np}$, because each of the n processes running on a host can receive messages from any of the other p processes in its application's process group. This solution, of course, reduces the buffer size for each process thus reducing the maximum bandwidth that can be achieved by an application running on such a cluster computer.

In this paper we study an alternative approach, that allocates the full send and receive buffers to the currently running process, so as to achieve maximal bandwidth. The communication state of other processes is stored temporarily in pageable buffers residing in each process's virtual memory. This is enabled by the use of gang scheduling on our multiprogrammed cluster: with gang scheduling, it is guaranteed that all the processes of a parallel application run in the same time quantum, and are dormant during all other quantums. Because of that, we know that no packets are sent to a process outside its time quantum (we do not allow inter-application communication). This feature allows us to switch the running process's communication buffer, along with its context. When a context switch is initiated by the scheduling mechanism, the physical buffer is copied into the virtual one, that can be paged, and the next process's buffers are copied from virtual space into the physical space for the duration of the time quantum. Since cluster computers using a gang scheduling scheme use a relatively large time quantum (measured in seconds or even minutes), and the communication buffers size is measured in megabytes, the overhead incurred by the buffer copying algorithm does not affect performance, as we shall see in Section 4.2.

2 System Background

2.1 The ParPar Cluster

The basic design of our cluster system, the ParPar [4], is simple. The hardware consists of 17 Pentium-Pro computers, running BSDI 3.1, connected by a control network which is a 10MB switched Ethernet, and a dedicated data network — a 1.28GB Myrinet [1]. The data network uses a modified version of the FM 2.0 library from the University of Illinois [12] which was adapted to our needs (more in Section 3). The software is a set of daemons: a master daemon — masterd — is run on one machine which is considered the host (or manager) of the cluster, and is not used by the user applications. Every other node runs a node daemon — noded — which manages the processes run on this node. These daemons communicate using the control network, while the data network is reserved for parallel applications communication.

When a user wishes to run a parallel application he contacts the masterd using a third program called the job representative — jobrep — which negotiates the loading of the applications with the masterd. The masterd then allocates nodes on which to run the application, and time slots during which to run the application, and notifies the nodeds running on the allocated nodes to run the application at a specific time slot.

The masterd manages the applications running on the cluster using a gang scheduling scheme. It maintains a matrix of size $n \times m$, where n is the number of time slots required, and m is the number of nodes (16 in our case). Each cell in the matrix represent a process of a specific parallel application associated with a physical node. This way several parallel applications can run in the same slot, as long as the sum of nodes they require does not exceed the total number of nodes. The mapping of applications into the matrix is based on the DHC scheme [5].

The masterd switches between time slots in a round-robin manner. Whenever a time quantum is finished, the masterd notifies the nodeds to switch to the application in the next time slot. This is done by a broadcast message [9]. The nodeds then start the three stage context switch procedure: they first flush the network, by signaling the Myrinet card to enter a context switch state. They then switch the context data and the communication buffers. Finally the nodeds signal the Myrinet card to return to normal communication status. This is explained in detail in Section 3.2.3.

2.2 The Fast Messages (FM) Library.

The FM library was developed by Prof. Andrew Chien and his team at the University of Illinois at Urbana-Champaign [12], as part of the High Performance Virtual Machine (HPVM) project. The library offers high speed communication over Myricom's Myrinet SAN network.

FM is composed of the following components:

- A library that is linked to user applications and contains an initialization routine and the basic routines for sending and receiving messages.

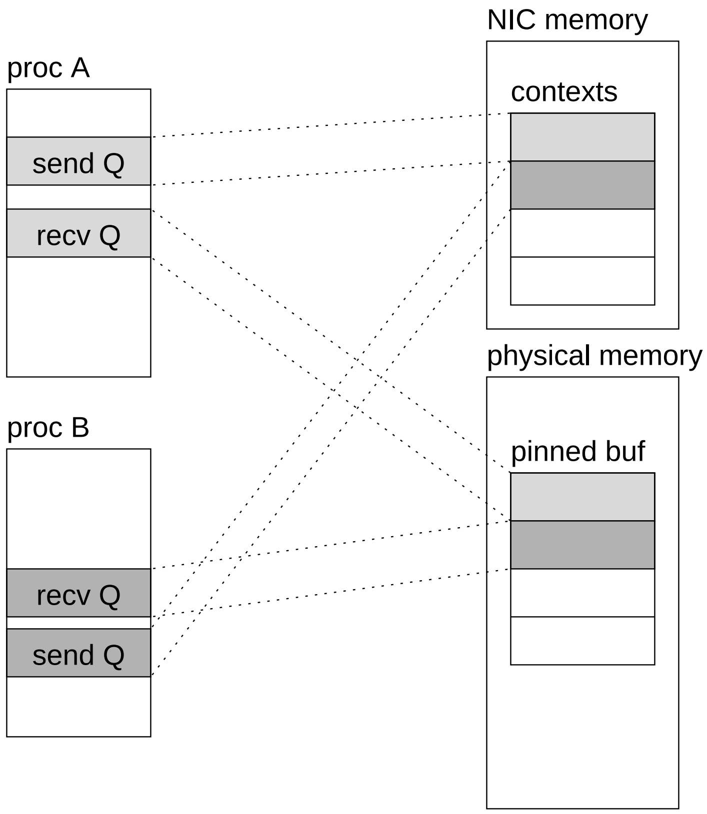

Figure 2.1: FM supports multiprogramming by dividing the buffer space.

- A control program that is executed on the LANai processor that resides on the Myrinet card.
- A global resource manager (GRM) daemon, that is responsible for assigning job and process IDs.
- A local context manager (CM) daemon on each node, responsible for the management of communication contexts for the processes on that node.

When a process that wishes to use FM starts running, it must first initialize the library using a function called FM_initialize. This includes contacting the GRM in order to perform the mapping from a job name (which is hardcoded) into a job ID (which is dynamically allocated and guaranteed to be unique). The process registers itself with the GRM, and receives the job ID and its rank in the job.

Each process that uses FM must have an FM context allocated for it on the Myrinet card, so the LANai can accept packets intended for that process, and to send the process' packets. Part of the initialization is therefore to contact the CM on a well known port, and register using the job ID and rank (received from the GRM). The CM allocates a context dedicated for this process, for as long as it runs. The context data includes the job ID the process belongs to, its rank in the job, and some sending counters, a dedicated process send queue which resides on the Myrinet card itself, and a dedicated receive queue which resides in the host RAM as a pinned DMA buffer. When using multiple contexts (that is, when supporting multiprogramming on each node) FM divides the available space on both the Myrinet card and the DMA buffer (which is pre-allocated by the driver during computer bootup) equally between the different contexts (Figure 2.1). This division is based on the fixed maximal number of contexts to be supported, and not adapted according to the number that are currently active.

The sending and receiving is done by a programmable general purpose CPU that resides on the Myrinet card: the LANai. The LANai is a dual context processor: one context is regarded as the user context, which is interruptible, and the other is considered the system context. Whenever the processor receives an interrupt it transfers control to the system context, which is not interruptible. FM runs a special program on this CPU that uses both contexts — The system context is used as the receiving context, and the user context is used as the sending context. Most of the time is spent in the send context, which keeps scanning the different processes' send queues for new packets to send, and sends them whenever one is available. Whenever a packet arrives, the processor receives an interrupt, transferring control to the receiving context, which consumes the packet from the network, identifies its type and destination and DMAs it to the target process's receive queue on the host. When the receive context finishes consuming all packet available on the network, it returns control to the send context.

FM's flow control algorithm is quite straight forward: each process is given a number of credits for sending packets to all other nodes in the system, where each credit represents one packet the process can send to a node. This means that space is available for this packet in the other node's receive queue. Each process manages two credit counters for each other node in the system — one counter marks the number of packets the process can send to that node, and the other the number of packets that can be received from that node. The credit is refilled when a refill message is sent between any two nodes. Such a message sent from node a to node b contains the number of packet from b that were consumed by a since the last refill message. Such a message is sent either when a notices b's credits fell below the low water mark level, or piggybacked whenever a sends a data packet to b.

The initial number of credits, which is also the maximal, is set according to the worst case scenario: assuming all nodes start sending packets to one node, what is the number of packets that can be received from each node so no packets will be lost. This number C_0 is determined by the formula $C_0 = \frac{B_r^{'}}{np}$, where $B_r^{'}$ is the size of the receive queue allocated for the process in packets, n is the maximum number of FM processes that can run on a single host, and p is the number of processors (hence the number of processes that can send to a single node is the overall number of processes in the system). Now, we must remember that the buffer size allocated for a process is the global buffer, divided by the number of processes in the host. This means that $B_r^{'} = \frac{B_r}{n}$, so the actual formula is $C_0 = \frac{B_r}{n^2 p}$, and as we can see there is an inverse square ratio between the number of contexts and the number of credits. This sharp reduction in the number of credits when the number of contexts is increased leads to a sharp degradation in the achievable bandwidth, as we shall see in Figure 4.1. Also note that because of this credit scheme and the credit refill technique, a single packet loss can mess up the credit counters and the entire flow control algorithm. FM does not have a retransmission mechanism. Since it uses a Small Area Network (SAN) it assumes an insignificant error rate, thus no retransmission will ever be required.

3 Integrating FM with ParPar

Our main goal in the integration was eliminating FM's GRM and CM daemons, since the ParPar already has its own daemons — both a global one (the masterd) and another one per node (the noded). All the functions fulfilled by the GRM are already fulfilled by the masterd. Since the masterd allocates nodes and IDs for new jobs, it has the same tables that are maintained by the GRM. Also, this data is known by the noded prior to execution, so there is actually no need to perform additional costly communication operations when a process is started. Since the functions accomplished by the CM are unique to FM, they are, of course, not accomplished already by the ParPar. However, they can easily be incorporated into the existing noded, without the overhead of an independent process.

3.1 General Network Management Library

The duplication of effort between FM and ParPar, as described above, is not unique to these systems. Most cluster management systems employ a similar structure of one central daemon and a set of per-node daemons (e.g. [11][7][14]). However, there is no agreed interface by which the required information can be passed to a communication system. Communication subsystems therefore have to find the required information by themselves. This is often achieved by constructing a rudimentary cluster management system whose sole purpose is to spawn the application processes on the various nodes and provide them with the required information, instead of using a more sophisticated external cluster management system.

One approach to avoid such duplication is to put the application and its communication system in the center, and have it contact an available cluster management system in order to allocate nodes and spawn processes [13]. However, this complicates the application (or at least the communication subsystem) by forcing it to deal with management issues — for example, what should it do if the requested number of nodes are not available? We prefer the opposite approach, in which allocation is first done by the cluster management system, which then calls a communication management library and provides it with the required information. Our goal when designing the interface for the network management library was therefore to design an abstract interface, that is independent of the cluster computer that uses it, and also of the actual communications library used. This interface gives the cluster programmer the ability to initialize and control the data network and allocate process resources with no knowledge of its actual media. Of course there is a need to adapt the library itself to each specific environment.

The interface can be divided into three parts: initialization, process control, and context switching (the complete prototypes for the functions can be found in Appendix A).

3.1.1 Initialization and maintenance

These functions let the cluster management system provide the communication library with important information regarding the configuration of the cluster as a whole. This is done during the initialization process, and when dynamic changes in the topology occur.

The functions in this section are:

- COMM_init_node - called when the daemon is started. The intention of this function is to initialize all the global, one time parameters, such as topology graph, initialize network interface, global counters etc.

- COMM_add_node - add another node to the topology. If a new node joins the cluster at runtime, this function allows the library to update its topology data.

- COMM_remove_node - remove a node from the topology. Whenever a node is killed or crashes, this function lets the other nodes update their known network topology.

Two more features that might enhance this interface section's capabilities are dividing the initialization process into two stages, to allow the network time to stabilize, and to separate the node initialization stage from the joining in stage, when the node joins all the group protocols. The other feature is some dynamic control over links, and not just over nodes — give the programmer the ability to change the status or capacity of a specific link in the topology. However, these features are not included in the current version.

3.1.2 Process control

This section gives us control over the network resources associated with a process that is part of a parallel job. These resources are send and receive queues, flow control counters etc.

The functions in this section are:

- COMM_init_job - allocate the resources needed for a new job, such as buffers, counters, etc. This function also builds the global variables that contain all the data required by the application to communicate using the data network.

- COMM_end_job - release whatever resources that were associated with the terminated job.

- COMM_change_slot - changes a process's slot in the table. By dictating the slot, the cluster management software and the communication library identify jobs using the same IDs, saving a level of indirection.

3.1.3 Context switch control

This section was the main focus of our research — it notifies the communications library of a context switch, so that the communications library will switch communication buffers and provide the new current process with all the physical resources. For the sake of flexibility, we looked at a network context switch as a three part sequence: flushing the network and halting communication, switching the contexts and buffers, and resuming communication and reflooding the network.

The functions in this section are:

- COMM_halt_network - halts all communication and flushes the network. This function initiates a flushing sequence on the network. It assumes the process itself was already stopped, and therefore will not produce any new messages. After this function returns, the network is declared stopped, and no communication is expected.
- COMM_context_switch - the actual switching of contexts and buffers. Saves the previous process's context, send queue, and receive queue into the process table, and reloads the next process's context and queues. After this function returns the node is ready for reflooding.
- COMM_release_network - resume communication. Release the new time slot's process so they can resume sending and receiving messages.

3.2 Implementing the Abstract Interface for FM

Although the interface was designed as a general purpose one, one goal was to integrate the FM into the ParPar system. The implementation for FM was quite straight forward. It consists of the following components:

- A library that is linked to user applications and contains an initialization routine and the basic routines for sending and receiving messages. This is essentially the same library as in the original FM — only the initialization function was modified, as described below.
- A control program that is executed on the LANai processor that resides on the Myrinet card. This is again essentially the same as in the original FM, with some additions to implement network flushing.
- A new library which we call "glueFM" that is linked with the noded. This library, composed of the functions defined above in Section 3.1, provides the functionality that was originally contained in the CM, and the new functions that we have defined (e.g. for context switching).

3.2.1 Initialization functions

Since we already had per host daemons running, the initialization part was implemented by simply copying the code from the CM into the COMM_init_node function of the library. This function is called when the noded is initiated, to load the control program into the LANai and initialize the topology and routing tables.

It gets this data from the masterd and/or from an FM configuration file that is NFS-mounted on all nodes.

3.2.2 Process control functions

The process control section was a little more complex. When an FM process starts running it contacts both the GRM and the CM to obtain its IDs and context. This is done using a three stage protocol, that also maintains synchronization between the running processes. This way no process can start sending to a process that has not yet started running, and no packets are lost. However, in the ParPar the nodeds have all the IDs prior to executing the FM process. These IDs are transferred to the process using environment variables. This mechanism replaces the first functionality of the original FM GRM-CM three stage initialization protocol, but leaves us with a synchronization problem: since the nodes are not synchronized, one node might execute the FM process before the others. Because this process knows the *"rank to node"* mappings prior to its run, it can start sending messages to other processes before they are executed. The LANai on the destination nodes will not know what to do with these messages and will drop them, leading to a loss of credits.

Our solution to this problem is divided into two parts (Figure 3.1). First, we separate the process's readiness for sending messages from its readiness for receiving messages. Even before the process is forked, the noded calls the COMM_init_job function. This updates the data in its context on the Myrinet card telling the LANai about its job ID and rank. If any messages now arrive, the LANai will receive them and store them in the receive queue (which is in physical memory). This can be done even if the process has not mapped it into its virtual address space yet.

At the same time, the noded notifies the masterd that the process has been created successfully. The masterd collects these notifications, and when all of them arrives, it notifies the nodeds. This provides a global synchronization point — when the notification arrives, the noded knows that all the processes in the job are up, and all the Myrinet cards are ready to accept messages. The noded forwards this information to the process by writing a single byte on a pipe that was created before the process was forked.

The process, for its part, calls the FM_initialize function that is part of the FM library with which it is linked (if the process uses a higher level communication system, such as MPI, it calls MPI_initialize, and MPI_initialize calls FM_initialize). We modified FM_initialize to obtain the data it needs (such as its rank in the job and its context on the LANai) from special environment variables that are set up in advance by the noded, instead of trying to get them from the GRM and CM. The actual format of these environment variables is set by

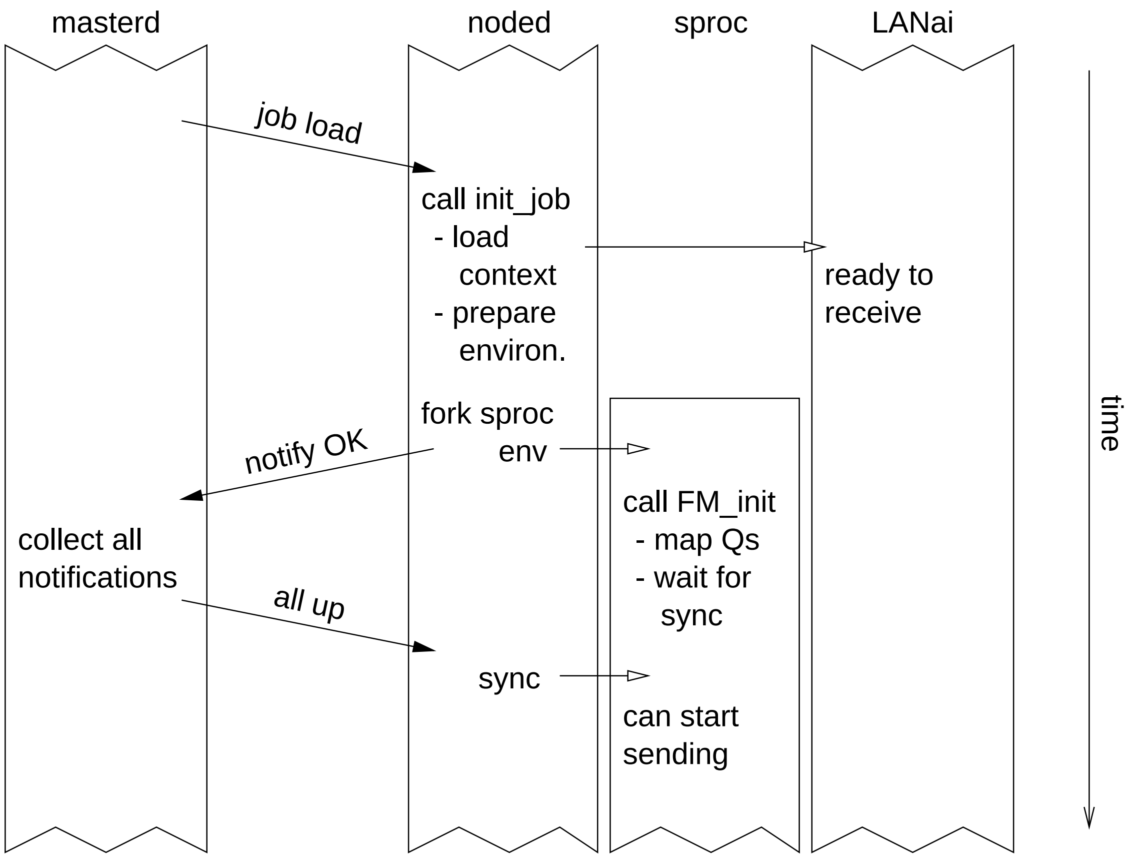

Figure 3.1: Procedure for initializing communication for a new job.

the COMM_init_job function; the noded just transfers them to the environment of the newly forked process. The function then opens the LANai and maps the send queue and receive queue into the process's address space. Finally, it waits for the global synchronization signal by trying to read a single byte from the pipe with the noded (the pipe's file descriptor is also passed in an environment variable). FM_initialize terminates after this synchronization is done, and the process can now start sending messages to other processes.

3.2.3 Context switch control functions

The implementation of the functions in this section was the main focus of our research. The implementation involves the LANai itself in the context switching, and by doing this, synchronizes the communication between the different processes, so only the running jobs will communicate during a time quantum. This feature frees us from the need to maintain different send and receive queues on each host, because only one is active at any given time, while the others are dormant. This, of course, adjusts the flow control algorithm for best performance and prevents unrelated processes' packets from entering the network, thus increasing the bandwidth.

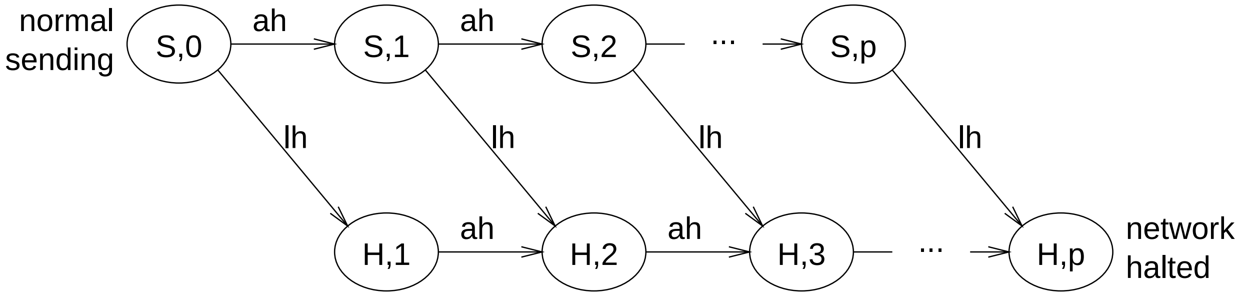

Figure 3.2: state transitions during the network flushing algorithm. Each state is marked by either S or H, indicating whether the node is still sending or halted. In addition it has a number indicating the total number of halted nodes we know of. An arriving halt message causes an "ah" transition and increments this number. A local halt transition ("lh") is caused by the noded.

As mentioned earlier, we separated the context switch into three stages - flushing the network, replacing the buffers, and reflooding the network. This was necessary because of the no-retransmission policy used by FM.

In order to prevent packet loss, we had to make sure there were no packets in transit when we switch buffers, so no packet will reach its destination node and be rejected because the target already switched contexts. This is done by the COMM_halt_network function, which ensures that the network is flushed, meaning that sending has been stopped at a packet boundary, and no incoming packets are expected from other nodes.

COMM_halt_network is called by the noded upon receipt of a context switch message from the masterd, after blocking the process by sending it a SIGSTOP. Therefore it is already assured that the process will not produce any more packets. However, some packets may be stored already in the send queue, and one may even be in the process of being injected into the network. The noded tells the LANai control program to stop sending additional messages by setting a flag in the LANai's memory. This has no effect on the current send, if one is in progress, but prevents additional packets from being sent. In addition, the LANai broadcasts a halt message to all other nodes, informing them that it will not send any more packets. Due to the FIFO quality of the Myrinet and the fact that FM uses a single precomputed route between each pair of nodes, this will indeed arrive after all previous packets. As the Myrinet hardware does not support broadcast, the broadcast is implemented by a serial loop. Note that these messages are sent between the Myrinet cards only, with no interaction with the host processor. They use specially tagged control packets, to distinguish them from the data packets. As they are just counted they do not need to be stored in buffers and do not require credits.

As the nodes are not fully synchronized, the different LANais will stop sending user packets and broadcast the halt message at different times. Thus a certain LANai may receive a halt message before it was notified by its noded that it should stop transmitting packets and enter the network flush phase. Flushing is therefore composed of two independent things:

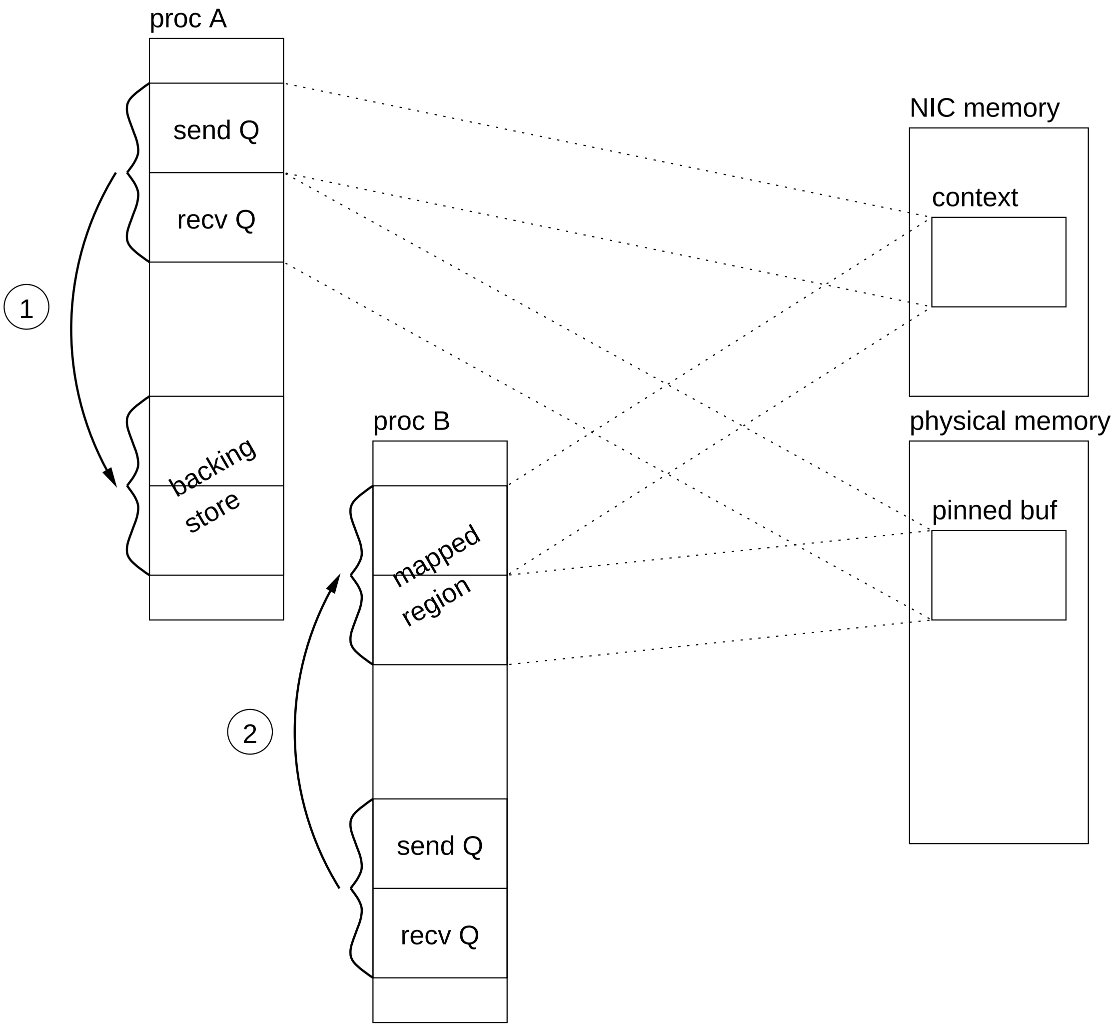

Figure 3.3: Copying buffers to switch from process A to process B.

one is the stopping of sending and the broadcast of the halt message, and the other is the collection of halt messages from all other nodes. The local halt can be interleaved with the collection of incoming halts in an arbitrary way. This is shown by the state transition graph in figure 3.2.

Because Myrinet has a FIFO quality, we know that all data packets that were sent before the halt stage have reached their destination after each node got a control halt packet from each other node. Even though it seems wasteful to use an all-to-all pattern, we found that this stage it several order of magnitudes shorter than the next stage of the actual buffer switch.

The next stage is the actual buffer switch. Since FM uses a fixed send queue on the Myrinet card and a receive queue as a DMA buffer pinned on the host, the buffer switch cannot be accomplished using a simple pointer swapping. Instead, it is necessary to copy the running queues into a backing store, and copy the new context's queues from its backing store. Also, the send queues resides on the Myrinet card itself, so its access time is much longer than a normal RAM access time, which is the case with the receive queue. This means that copying such buffers is quite lengthy. However, we discovered that usually the queues are quite empty: while the receive queue size in packets is 668 it usually did not contain more than a few dozen packets, and the send queue did not even exceed a few single packets out of 252. This phenomenon can be explained by the fact that the send queue is filled by the host processor, which also takes care of the data processing and other processes, while it is emptied by the LANai, which only handles the sending and receiving. Because of this, the host processor (a 200 MHz Pentium-Pro) cannot generate messages fast enough to fill the queue. The receive queue emptiness can explained by FM's window-based flow control mechanism: the large receive queue provides the sender with enough credits so that it can send continuously, without waiting for acknowledgements. Normally, packets are removed from the queue at a rate similar to the rate at which they arrive, and the queue stays relatively empty. The queue fills up only under special circumstances when the processor does not manage to remove packets fast enough. When this occurs, the receive queue serves to buffer the incoming packets until the sender runs out of credits and stops transmitting additional packets, but in practice, this did not happen during the measurements.

The third a final stage of the context switch is the reflooding stage. This stage is implemented using an identical protocol to the one used in the first stage. Each node broadcasts to all other nodes that it is ready to receive messages for the new context. When a node receives messages from all other nodes that they are ready to receive, it can resume sending safely. Again, the reflooding protocol takes place between the Myrinet cards with no interference from the host processor. The noded simply initiates it by changing a variable located on the Myrinet card. This signals the LANai to start the flood protocol. The communication itself is done by using specially tagged control packets, to distinguish them from the data packets.

This context switch mechanism was found to be robust, and withstood thorough testing without packet loss.

3.3 Flow Control Algorithm

The only change we had to make to the FM library itself (as oppose to the other changes that were made to the accompanying daemons) was adjusting the flow control parameters. Let us first recall maximal number of credits calculation: we have to divide the buffer allocated for the process between all the processes that can send it, or simply $C_0 = \frac{B_r'}{np}$, where B_r' is the size of the receive queue allocated for the process in packets, n is the maximum FM processes that can run on a single host, and p is the number of processors. In the original FM library, the total receive buffer is equally divided among all the processes running on a single host, so $B_r' = \frac{B_r}{n}$. Using this scheme we had to set the number of contexts (number of FM processes running on the same host) to be the size of the time slot table — the maximum number of parallel application that can be run alternately using the gang scheduling mechanism.

Since we are using a multiprogrammed parallel computer with gang scheduling scheme, all the processes in a parallel job have to run simultaneously. This means that only one process per job can run on a single host, so the maximum number of processes that can send to a specific process is reduced to p (instead of np), or more formally, $C_0 = \frac{B_r^{'}}{p}$. Now, since only one process uses the global receive buffer for each host at any given time, and the buffer is saved each context switch, there is no need to divide the global receive buffer by the number of processes running on a host to form an exclusive receive buffer for each process. Instead, this allows us to increase the receive buffer used by each process to the entire global buffer, or $B_r^{'} = B_r$. Overall, these adjustments increased the maximal credit number by a factor of n^2 to $C_0 = \frac{B_r}{p}$. This increase allows better utilization of the receive queue, without increasing the buffer itself.

4 Performance Results

We measured two aspects of the algorithm: the improvement in the overall bandwidth delivered by the system to the parallel jobs running in the gang scheduling scheme, and the overhead incurred by the context switch itself — the time it took to perform each stage of the network flush and buffer switch procedure.

4.1 Effects on Bandwidth

We first measured the overall bandwidth available in the system. Our basic bandwidth benchmark was based on the bandwidth benchmark that comes as part of the FM distribution. The benchmark itself is a simple point-to-point measurement: a parallel application which consists of two processes, a sender and a receiver. When run, the sender starts sending a given number of messages of a specific size. After all the messages are received by the receiver, it sends a finish messages to the sender and exits. When the sender receives the finish message it times it and calculated the bandwidth. Although the finish message incurs some overhead which hinders the test's accuracy, this can be offset by using a large enough number of messages. We used 500,000 for small messages and 100,000 for large ones.

We ran the benchmark as a single application on the ParPar system, so no context switches occurred. This gave the overall bandwidth available using the buffer division in a system that can run several applications alternately. The results for running the benchmark using the original FM scheme can be seen in figure 4.1. As we can see the bandwidth decreases sharply when increasing the number of contexts (or dividing the buffer size). No communication is even possible for as few as 8 contexts, or 8 parallel applications running alternately using our gang scheduling scheme. For small message sizes, a full creadit is used even if only part of each packet is used, so the system becomes unusable with even less contexts. The standard FM distribution actually supports only 2 contexts per node, so it does not reveal these problems. The results also indicate that about 256KB of memory on the NIC suffices for adequate performance; hence as the available memory grows, more contexts can be supported.

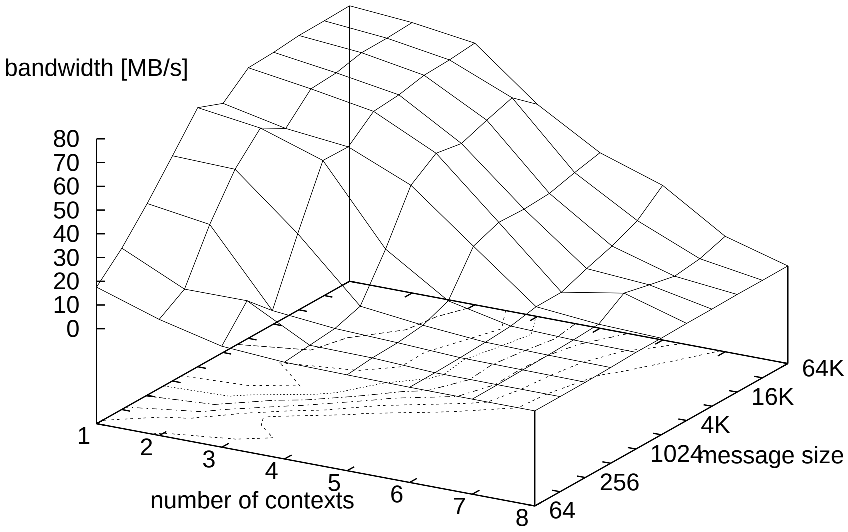

Figure 4.1: Bandwidth measurement for FM as a function of message size and the number of contexts, using the original FM buffer division.

We then measured the available point to point bandwidth when using our scheme. As we recall, we do not divide the global buffer among the running processes, but rather switch it whenever the gang scheduling controller initiates a global context switch. To measure the overall bandwidth available by the system in this method, we had to run several benchmark applications simultaneously, and alternate between them using the gang scheduling algorithm, running with a time quantum of 3 seconds. To obtain the overall bandwidth achievable in the system, we multiplied the average bandwidth achieved by the benchmark applications, by the number of applications running simultaneously. This compensated for the fact that each application was effectively using only a fraction of it's elapsed runtime. Even though this measurement incurs the overhead of the buffer switching, the overall available bandwidth is independent of the number of applications running in the system, and maintains a fairly constant level as seen in figure 4.2. This shows conclusively that when using the gang scheduling + buffer switch method, the presence of multiple applications running does not impair the overall bandwidth available in the system, so a multiprogrammed computer cluster can be more effectively utilized, using our approach.

4.2 Overheads

When assessing our algorithm, we had to measure the overhead incurred by the buffer switch added to every gang scheduling context switch.

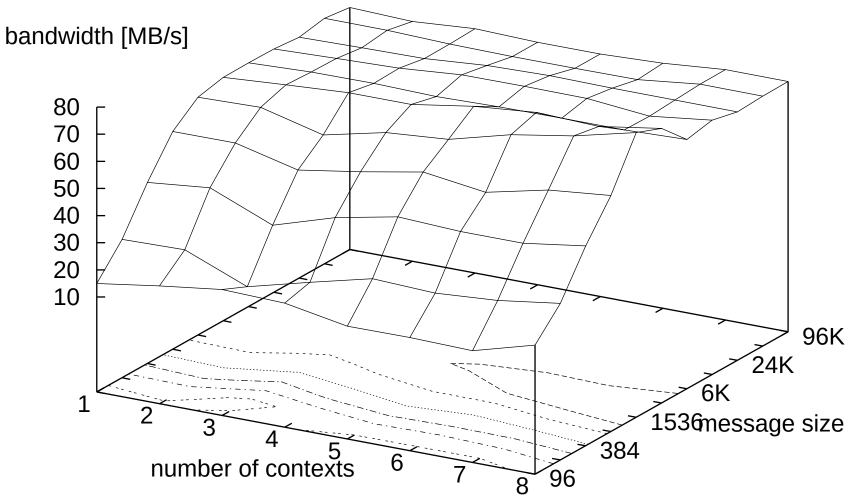

Figure 4.2: Total bandwidth as a function of message size and number of jobs, using the buffer switching scheme.

4.2.1 Technical data

As mentioned before, our cluster consists of 17 Pentium-Pro 200 computers (one serves as a master, and 16 nodes), running BSDI 3.1. Each computer has a Myrinet card with a LANai 4.3 processor, and 512KB RAM.

The receive buffer is a 1MB pinned down DMA buffer, located in the computer's RAM, and allocated by the Myrinet driver at boot time. The send buffer is ˜400KB in size, and is located on the Myrinet card's own RAM. With FM's packet size of 1560 bytes, it means the receive buffer is of 668 packets in size, and the send buffer is of 252 packets in size. Since the send buffer is mainly written to and not read from, FM's designers used an optimization available on the P6 processors family (Pentium-Pro and up) called *'write-combining'* [3]. This optimization, which was originally developed for graphic controllers, changes a memory region accessing policy to be uncacheable, and all writes to that region are then accumulated in an on-chip cache-line-sized write buffer and written to memory in one bus transaction. This optimization accelerates memory write speed immensely, while decelerating memory reads drastically. On our hardware, while regular memory accesses were measured at ˜45MB/s, write-combining memory read bandwidth was measured to be as low as ˜14MB/s, whereas write-combining memory write bandwidth rocketed to ˜80MB/s. This caused that even though the receive buffer is more than twice the send buffer's size, the time consuming part of the buffer switch was replacing the send buffer.

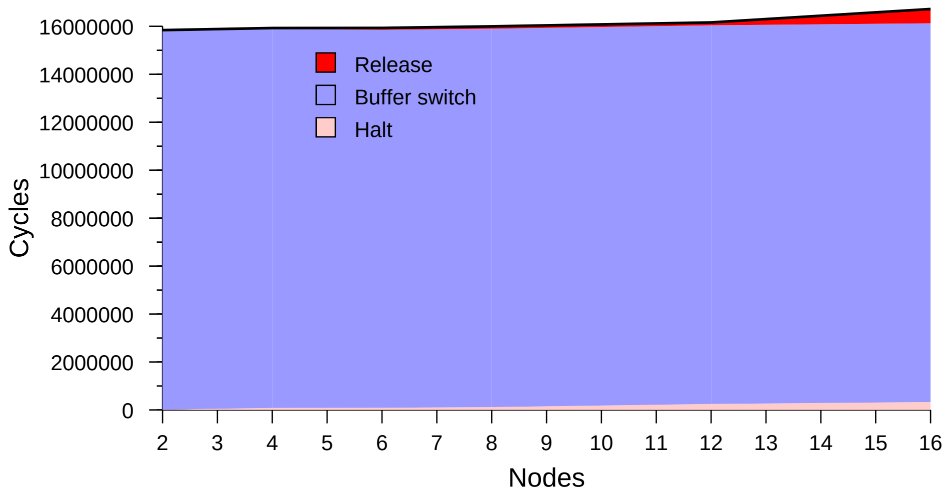

Figure 4.3: Time measurements for the buffer switch algorithm, in cycles.

4.2.2 Measurements description

To measure the context switch overhead we used an all-to-all benchmark, that will stress the buffers during the test. We measured each of the three stages of the buffer switch algorithm. As we can see in figure 4.3, the vast majority of the time consumed by the switch was spent on the second stage — the buffer switch itself. We can also see that the flush and reflooding stages consume more time as more nodes are involved. This effect is expected since these stages involve a global protocol between unsynchronized computers. The buffer switch time, on the other hand, does not depend on the number of nodes in the system because it is a local procedure, and does not involve any external nodes.

These measurements lead us to inspect if the copying the entire buffers is really necessary — how many valid packets are waiting in the buffers when the buffer switch occurs, and can we simply copy only parts of the buffers to save cycles? We saved the number of packets present in the buffers on every buffer switch, and the results can be seen in figure 4.4. These result shows us that the buffers a generally quite empty. The increase in the number of valid packets in the receive buffer can be blamed on the all-to-all communication pattern. The increase in messages sent does not fill the send buffer (an average of ˜0.1 packets - the graph shows the ˜1.1 packets copied on average, because one packet is copied anyway, to prevent partial packet loss) because the LANai processor's only job is to empty it to the network. However, the host processor cannot keep up with the bursts of incoming packets, and the receive buffer fills up to some degree.

This finding lead us to improve our buffer switch algorithm: go through the buffers and only copy the valid packets. We measured the improved algorithm, whose results can be seen in figure 4.5. It can be seen that the time needed to switch the buffers is reduced dramatically,

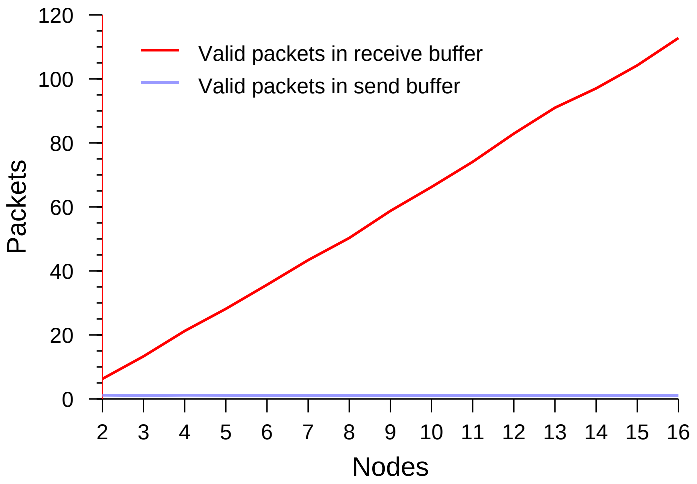

Figure 4.4: Number of valid packets in the buffers during buffer switching.

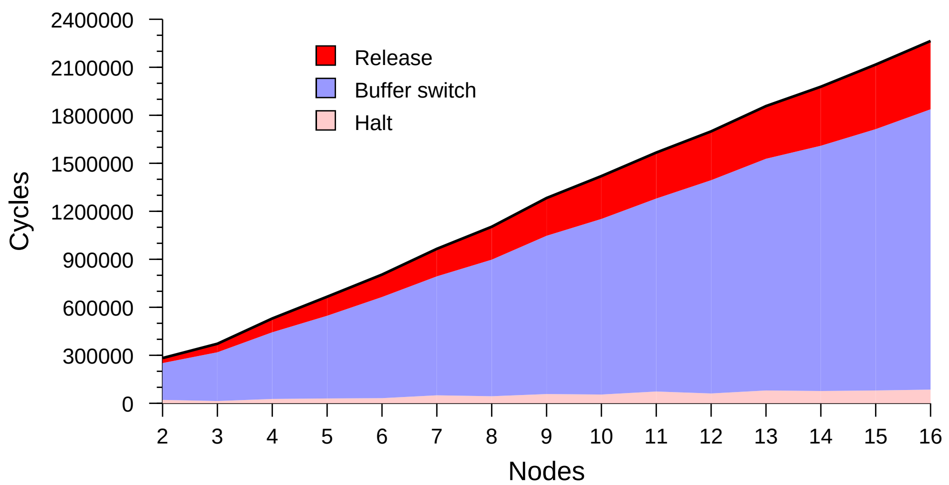

Figure 4.5: Time measurements for the improved buffer switch algorithm, in cycles.

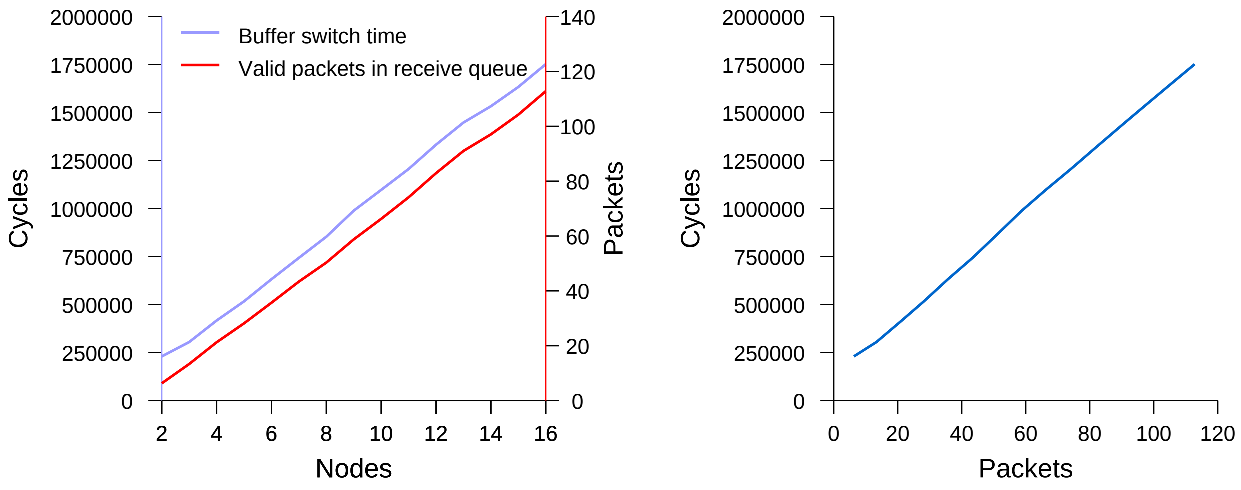

Figure 4.6: The correlation between the number of valid packets in the receive buffer and the buffer switch time

while the other stages were not affected. We wanted to verify the the dependency of the buffer switch time on the number of nodes corresponds to that of the number of valid packets in the queue. We checked both dependencies for linearity (figure 4.6, left), to find a correspondence between the two graphs. This correspondence was also verified by comparing the growth ratio between the buffer switch time and number of valid packets in the receive buffer (figure 4.6, right). Again, the ratio showed a fixed growth rate that indicates a linear dependency between the two factors, that explains the growth in the buffer switch time.

These overhead measurements indicates that the overhead incurred by the buffer switch is negligible comparing to the long time quantum used in multiprogrammed gang scheduling machines (seconds or even minutes). This overhead was even further reduced (as shown in figure 4.7) when we searched the buffers and only copied the valid packets. Figure 4.7 also shows that when using a 200MHz Pentium-Pro and the improved buffer switch algorithm, the buffer switch takes less than 12.5msecs (2,500,000 cycles). We ran our overhead measurements using a 1 second time quantum, so this overhead is less than 1.25%! Even when using the full buffer switch the time is less than 85msecs (17,000,000 cycles), an overhead which is tolerable even for such a short quantum.

5 Related Work

Other clusters using user-level communication have also dealt with the interaction between the user-level communication and the process scheduling.

FM specifically was used as the platform to investigate dynamic coscheduling [15]. The idea here is that instead of using gang scheduling, processes will be co-scheduled on the differ-

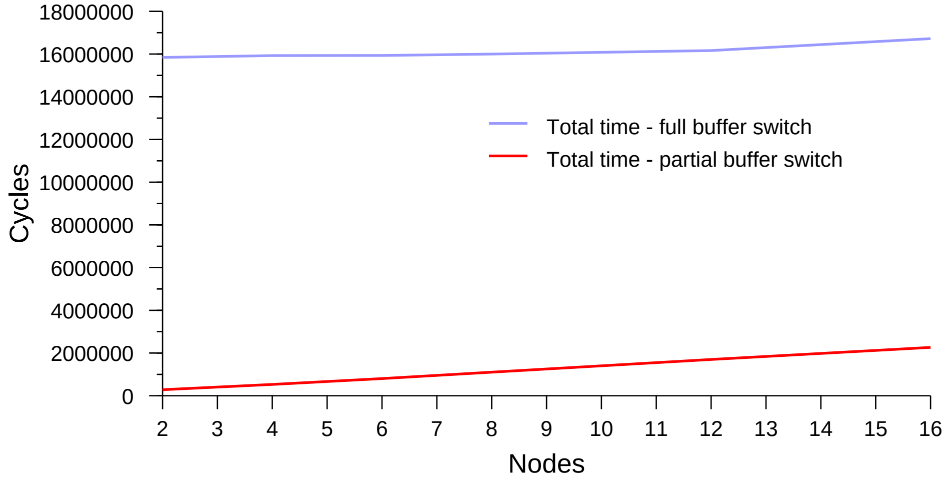

Figure 4.7: Comparison between the full buffer copy and the partial buffer copy times

ent nodes only if this is warranted by the interactions between them. This was implemented based on a modification to FM so that incoming messages would trigger the scheduling of the processes to which they are destined. However, this was done with version 1 of FM, which only supported a single full-size context (the competing workload consisted of local sequential processes that do not communicate). They therefore did not face the bandwidth problems that occur when multiple contexts are used and the size of each one is reduced.

The SHARE scheduler for the IBM SP2 switches communication buffers as we do, citing the problem of having to pin too much memory as the reason [6]. However, their implementation is quite different. First, all coordination in that system is derived from the use of synchronized clocks. The nodes do not interact in order to synchronize, and do not receive broadcasts from a central controller. In particular, the network is not flushed as part of a context switch, and nodes do not know exactly when other nodes complete their switching. Therefore it may happen that a node receives a packet destined for a process that is no longer running. This is handled by comparing an ID carried in the packet with an ID for the current process stored on the NIC, and discarding the packet if it does not fit. It is assumed that higher-level software (e.g. MPI or TCP) will handle the retransmission needed to compensate for such lost packets.

Flushing the network as part of a context switch was pioneered by the CM-5 Connection Machine [10]. This implementation has the distinction of flushing messages that are in transit, and storing them on any node in the partition. When the job is re-scheduled, these messages are re-injected into the network to complete their trip. Flushing is also used in the SCore-D cluster, which uses the PM user-level communication library [8]. This most closely resembles our work, but again there are differences in the details of the implementation. Specifically, PM uses nack messages and resends when there is no space in the receive buffer, rather than relying on credits. Thus there is no need to send special control messages in order to flush the network: each node simply stops transmitting, and then waits until it receives acks or nacks for all outstanding packets.

Another similar idea lies at the basis of virtual networks [2]. In this project, the solution for the lack of space on the NIC is to cache active endpoints on the NIC, while moving inactive ones to backing store on the node computer. This approach is different from the others in that it does not create any linkage between the communication subsystem and the scheduling of communicating processes.

6 Conclusions

High-performance communication requires direct access to important resources such as the scarce on-board memory available on network interface cards. Therefore the implementation of such mechanisms should be coordinated with other resource management policies, and specifically, with process scheduling. We have shown that lack of coordination results in the need to split the scarce resources among competing processes, which leads to significant degradation in communication performance. Our solution is to use time slicing and switch the buffers from one process to the next. This is possible due to the use of gang scheduling, because then we are assured that when a certain process is descheduled (and looses access to the communication buffers) all its potential communication partners are also descheduled. While such switching incurs the overhead of copying the buffer contents, we showed that typically only a small portion of the buffer is actually occupied, so the overhead is actually rather small.

As part of this work we also defined an interface for the integration of cluster management systems with high-performance communication systems. An adoption of such an interface by multiple systems would help in the design and implementation of high-performance clusters, as it would allow each development group to concentrate on part of the problem, while being able to use complementary software developed by others. Without such an interface, each cluster system needs to include both cluster management and communication functions, with are incompatible with those of other systems, and cannot be interchanged.

Acknowledgements

This research was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities. We are grateful to Prof. Andrew Chien for providing a source license of FM which enabled this research.

A Abstract Network Control Interface

A.1 Types

- struct rank2nodeid_t {
  int nodeid;
  int rank;
  };

  This type is used to hold the map from the process's rank in the parallel job, to the physical node ID on which it is run. Used to pass this table when initializing a job, so the routing tables can be built.

- struct hostname2nodeid_t {
  struct in_addr node_ipaddr;
  int node_id;
  };

  Type used to map between a hostname (or simply its IP address), and the node ID. Since the node IDs are uniquely allocated by the masterd, this type is needed to transfer this mapping around (for more details of struct in_addr see [16]).

A.2 Macro Interface

- COMM_OK
- COMM_ERROR
- COMM_TIMEOUT

  The library's standard return values, which indicate if an operation succeeded, failed or timed out, respectively.

A.3 Network Maintainance Functions

- int COMM_init_node(int my_nodeid, hostname2nodeid_t *nodes, int numnodes)

  The intention of this function is to initialize all the global, one time parameters, such as topology graph, initialize network interface, global counters etc.

  Parameters:
  my_nodeid - the unique ID of the node the function is executed on (more precisely -

the node on which the daemon calling the function is running on). Required to locate this node in the topology map.
nodes - the mapping from the nodes' names (or IP addresses) to the system's unique node IDs. Since topology files are usually written using either node names or IP addresses, this map is required to build the routing tables.
numnodes - the number of nodes in the map, and in the system in general.

Return value:
Either COMM_OK on success, or COMM_ERROR on failure.

- int COMM_add_host(int nodeid)

  Add another node to the topology. If a new node joins the cluster at runtime, this function allows the library to update its topology data. The noded ID inserted must have been present in the map received by the initialization functio - COMM_init_node().

  Parameters:
  nodeid - the new node's ID.

  Return value:
  Either COMM_OK on success, or COMM_ERROR on failure.

- int COMM_remove_host(int nodeid)

  Remove a node from the topology. Whenever a node is killed or crashes, this function lets the other nodes update their known network topology.

  Parameters:
  nodeid - the removed node's ID.

  Return value:
  Either COMM_OK on success, or COMM_ERROR on failure.

A.4 Process Control Functions

- int COMM_init_job(int progid, int rank, int slotid, const rank2nodeid map[], int numprocs, char **pp_map_p, char **pp_comm_specific_p, int startrunning)

  Allocate and initialize all the resources needed for a new job, such as buffers, counters, etc. This function also builds the global variables that contain all the data required by the application to communicate using the data network.

Parameters:
progid - the ID of the parallel job to which this process belongs.
rank - this process's rank in the parallel job.
slotid - the time slot in which this job is run. This is required to associate the job with its communication buffers, which are accessed by time slot ID.
map - the mapping of the job's processes to physical nodes.
numprocs - the number of processes participating in this parallel job. This is also the number of element in map.
pp_map_p - a pointer to a character array, into which to write the mapping from the process's rank in the parallel job to the physical node on which it is run. This string is later transtferred to the library linked with the process inself via an environment variable. The string is written by the "glue" library to allow each implementation to have its own map format.
pp_comm_specific - a pointer to a character array that will hold any library specific variables which are needed by a process participating in a parallel job using the communication media used. This array will also be transferred to the process as an environment variable.
startrunning - a boolean paramter to indicate to the network control whether the parallel job is started queued, or running immediately.

Return value:
The function returns COMM_OK on success, or COMM_ERROR on failure (buffers too small, cannot initialize network, etc.)

- int COMM_end_job(int progid, int rank, int slotid)

  Release whatever resources that were associated with the terminated process whose rank is rank, and belongs to the parallel job progid.

  Parameters:
  progid - the parallel job to which the terminated process belongs.
  rank - the terminated process's rank within the parallel job.
  slotid - the time slot this parallel job is run in.

  Return value:
  The function returns COMM_OK on success, or COMM_ERROR on failure.

- void COMM_change_slot(int old_slot, int new_slot)

  Changes a process's slot in the table. By dictating the slot, the cluster management software and the communication library identify jobs using the same IDs, saving a level of indirection.

  Parameters:
  old_slot - the current time slot in which the job is run.
  new_slot - the new time slot in which the job should run.

  Return value:
  The function returns COMM_OK on success, or COMM_ERROR on failure.

A.5 Context Switch Management

- int COMM_halt_network(unsigned long haltid)

  Halt all communication and flush the network. This function initiates a flushing sequence on the network. It assumes the process itself was already stopped, and therefore will not produce any new messages. After this function returns, the network is declared stopped, and no communication is expected.

  Parameters:
  haltid - a unique ID for each network operation. Allows the programmer to verify if several halt operations received from different nodes match.

  Return value:
  Either COMM_OK on success, COMM_ERROR on failure or COMM_TIMEOUT if the halt operation timed out.

- int COMM_context_switch(int switch_to_slot)

  The actual switching of contexts and buffers. Saves the previous process's context, send queue, and receive queue into the process table, and reloads the next process's context and queues. After this function returns the network is ready for reflooding.

  Paramter:
  switch_to_slot - the next time slot to switch to.

  Return value:

Either COMM_OK if the operation was successful, or COMM_ERROR on failure.

- int COMM_release_network(unsigned long releaseid)

  Resume communication. Release the new time slot's process so they can resume sending and receiving messages.

  Parameter:
  releaseid - a unique ID for each network operation. Allows the programmer to verify if several halt operations received from different nodes match.

  Return value:
  Either COMM_OK on success, COMM_ERROR on failure or COMM_TIMEOUT if the release operation timed out.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K. Su, "*Myrinet: a gigabit-per-second local area network*". *IEEE Micro* **15(1)**, pp. 29–36, Feb 1995.

[2] B. N. Chun, A. M. Mainwaring, and D. E. Culler, "*Virtual network transport protocols for Myrinet*". *IEEE Micro* **18(1)**, pp. 53–63, Jan/Feb 1998.

[3] Intel Corp., *Write Combining Memory Implementation Guidelines.* Order number 244422-001, Nov 1998.

[4] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. A. Volovic, "*The ParPar system: a software MPP*". In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, R. Buyya (ed.), pp. 754–770, Prentice-Hall, 1999.

[5] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[6] H. Franke, P. Pattnaik, and L. Rudolph, "*Gang scheduling for highly efficient distributed multiprocessor systems*". In 6th *Symp. Frontiers Massively Parallel Comput.*, pp. 1–9, Oct 1996.

[7] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, "*GLUnix: a global layer Unix for a network of workstations*". *Software — Pract. & Exp.* **28(9)**, pp. 929–961, Jul 1998.

[8] A. Hori, H. Tezuka, and Y. Ishikawa, "*Overhead analysis of preemptive gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 217–230, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[9] A. Kavas, D. Er-El, and D. G. Feitelson, "*Using multicast to preload jobs on the ParPar cluster*". *Parallel Comput.*, 2000. (to appear).

[10] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". *J. Parallel & Distributed Comput.* **33(2)**, pp. 145–158, Mar 1996.

[11] B. C. Neuman and S. Rao, "*The prospero resource manager: a scalable framework for processor allocation in distributed systems*". *Concurrency — Pract. & Exp.* **6(4)**, pp. 339–355, Jun 1994.

[12] S. Pakin, V. Karamcheti, and A. A. Chien, "*Fast messages: efficient, portable communication for workstation clusters and MPPs*". *IEEE Concurrency* **5(2)**, pp. 60–73, Apr-Jun 1997.

[13] J. Pruyne and M. Livny, "*Interfacing Condor and PVM to harness the cycles of workstation clusters*". *Future Generation Comput. Syst.* **12(1)**, pp. 67–85, May 1996.

[14] S. H. Russ, K. Reece, J. Robinson, B. Meyers, R. Rajan, L. Rajagopalan, and C-H. Tan, "*Hector: an agent-based architecture for dynamic resource management*". *IEEE Concurrency* **7(2)**, pp. 47–55, Apr-Jun 1999.

[15] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "*Dynamic coscheduling on workstation clusters*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 231–256, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[16] W. R. Stevens, *Unix Network Programming.* Prentice Hall, 1990.