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 scarce resources that limits communication performance is buffer space on the network interface card. This becomes even worse when it is partitioned among several time-sliced processes. However, if gang scheduling is used, it is possible to swap buffer contents as part of the context switch, giving each job the full buffer space for the duration of its quantum. This does not suffer undue overhead, as the buffer space is mainly used to allow a larger flow-control window, and typically does not contain many packets that need to be stored.

1. Introduction

It is widely known that buffer sizes affect communication performance. The buffer size limits the size of both the receive queue and the send queue. These sizes dictate with the flow control window size — the number of messages that can be sent before an acknowledge reply is required.

However, there are limits on the usable buffer size. The total amount of physical memory is finite, and most of it has to be allocated to the application itself, not to communication buffers. Large communication buffers are especially troublesome, because they typically have to be pinned in order to support DMA, thus limiting the operating system's flexibility in memory management. Finally, some of the buffers are located on the network interface card itself, which has very limited space.

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. The simplest solution 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 sizes 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 among the running processes), and the effective receive buffer size will be $\frac{B_r}{np}$, 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 reduction in effective buffer size limits the maximum bandwidth that can be achieved.

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, which guarantees that all the processes of a parallel application run in the same time quantum, and are dormant during all other quantums. Thus no packets are sent to a process outside its time quantum (we do not allow inter-application communication). A context switch stores the contents of the communication buffers together with the process's regular context. Since clusters using gang scheduling 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 does not affect performance.

2. System Background

2.1. The ParPar Cluster

The configuration of our cluster system, the ParPar [4], is based on 17 Pentium-Pro computers, running BSDI 3.1. These are connected by a 10MB switched Ethernet that serves for control functions, and a 1.28GB Myrinet [1] for data communications. The Myrinet network interface cards have a LANai 4.3 processor and 512 KB RAM. Data communications use a modified version of the FM 2.0 library from the University of Illinois [10] (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 on this node. These daemons communicate using the control network, while the data network is reserved for parallel applications.

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 notifies the nodeds running on the allocated nodes to run the application. Allocation is based on a gang scheduling matrix with 16 columns (representing the 16 nodes) and n rows, where n is the number of time slots required. Each cell in the matrix represents 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 [8]. 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.

2.2. The Fast Messages (FM) Library

The FM system was developed at the University of Illinois as part of the High Performance Virtual Machine (HPVM) project [10]. The system offers high speed communication over Myricom's Myrinet SAN (System Area Network). It 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.
- 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 hard-coded) 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.

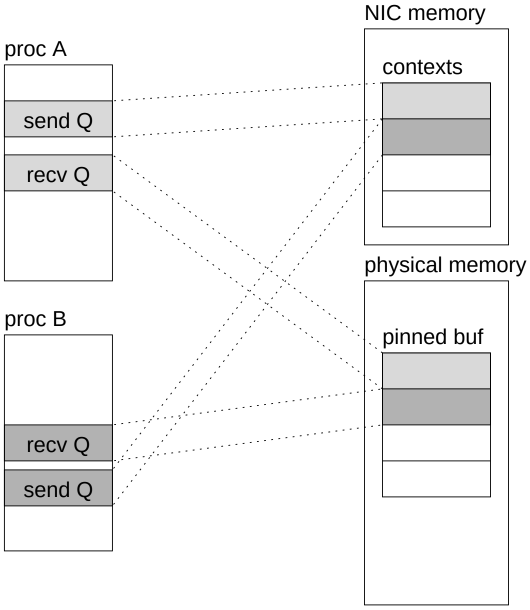

Figure 1. FM supports multiprogramming by dividing the buffer space.

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 send the process's 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, some sending counters, a dedicated send queue which resides on the Myrinet card itself, and a pointer to 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 of both the Myrinet card and the DMA buffer (which is pre-allocated by the driver during computer bootup) equally between the different contexts (Figure 1). This division is based on the fixed maximal number of contexts to be supported, and is not adapted according to the number that are currently active.

The sending and receiving is done by the LANai, which is a programmable general purpose CPU that resides on the Myrinet card. This is a dual context processor. 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. When a packet arrives, the processor receives an interrupt, transferring control to the receiving context. The program in this context 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 packets available on the network, it returns control to the send context.

FM's flow control algorithm is quite straightforward. Each process is given a number of credits, each of which represents one packet the process can send to another 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 counts 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. Refill messages are 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. Given that the buffer size allocated for a process is the global buffer divided by the number of processes in the host, $B_r^{'} = \frac{B_r}{n}$, the actual formula is $C_0 = \frac{B_r}{n^2 p}$. Thus 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 5. 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, based on the assumption of an insignificant error rate on a SAN.

3. Integrating FM with ParPar

Our main goal in the integration was to eliminate FM's GRM and CM daemons, since ParPar already has its own daemons — both a global one (masterd) and another one per node (noded). All the functions fulfilled by the GRM are already fulfilled by the masterd. Moreover, the required job ID and rank are 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.

Initialization and maintenance:
COMM_init_node - initialize LANai, contexts, routing table
COMM_add_node - update topology
COMM_remove_node - update topology
Process control:
COMM_init_job - allocate context, prepare environment variables for FM_initialize
COMM_end_job - cleanup
Context switch control:
COMM_halt_network - stop sending and perform global network flush protocol
COMM_context_switch - swap buffers
COMM_release_network - synchronize and restart sending

Table 1. API of network management library.

3.1. General Network Management Library

Regrettably, there is no agreed interface by which the required information can be passed to a communication subsystem. One approach 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 [11]. 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 specific cluster management system and communications library. The implementation of the network management library, however, must obviously be adapted to each specific environment.

The interface can be divided into three parts: initialization, process control, and context switching. The functions are listed in Table 1 (the complete prototypes for the functions can be found in an extended version of this paper at http://www.cs.huji.ac.il/˜feit/gang_comm.ps.gz).

3.2. Implementing the Abstract Interface for FM

The integration of FM into the ParPar system 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 Table 1, provides the functionality that was originally contained in the CM, and the new functions that we have defined (e.g. for context switching).

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 initialized, 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.

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 ParPar the nodeds have all the IDs prior to executing the FM process, and this data is simply transferred to the process using environment variables. This leaves us with a synchronization problem, since the first node to come up may start sending messages to other processes before they are ready. 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 2). 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 allocates a context on the Myrinet card initializing it with the 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 this buffer into its virtual address space yet.

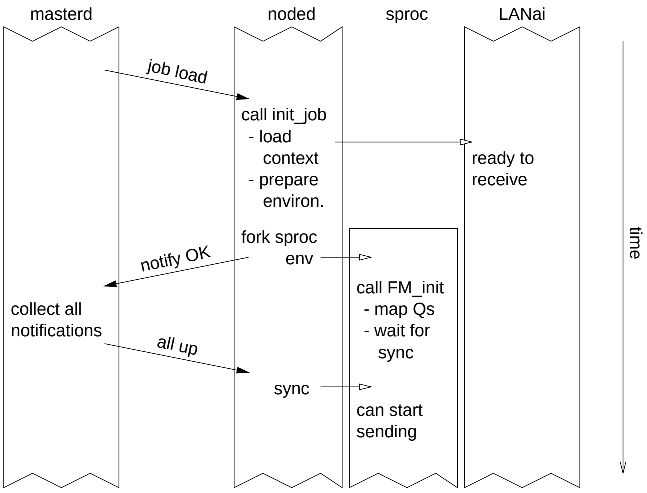

Figure 2. Procedure for initializing communication for a new job.

After forking, the noded notifies the masterd that the process has been created successfully. The masterd collects these notifications, and when all of them arrive, it notifies the nodeds. This provides a global synchronization point. 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 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.

Context switch control functions Upon receipt of a context switch message from the masterd, the noded blocks the

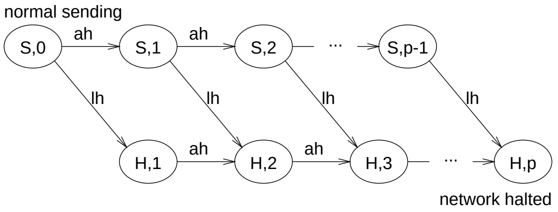

Figure 3. State transitions during the network flushing algorithm. States are marked with S (still sending) or H (halted), and a number indicating the total number of halted nodes we know of. An arriving halt message causes an "ah" transition. A local halt transition ("lh") is caused by the noded.

current process by sending it a SIGSTOP. At this point it is assured that the process will not produce any more packets, but some packets may be stored already in the send queue, and one may even be in the process of being injected into the network. To stop transmission on a packet boundary, the noded calls COMM_halt_network. This function sets a bit in the LANai's memory. The LANai control program checks this bit before sending each packet, so once it is set additional packets will not be sent.

Stopping transmissions is not enough — we must also ensure that no additional messages are expected to arrive. After stopping transmissions the LANai therefore 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: 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.

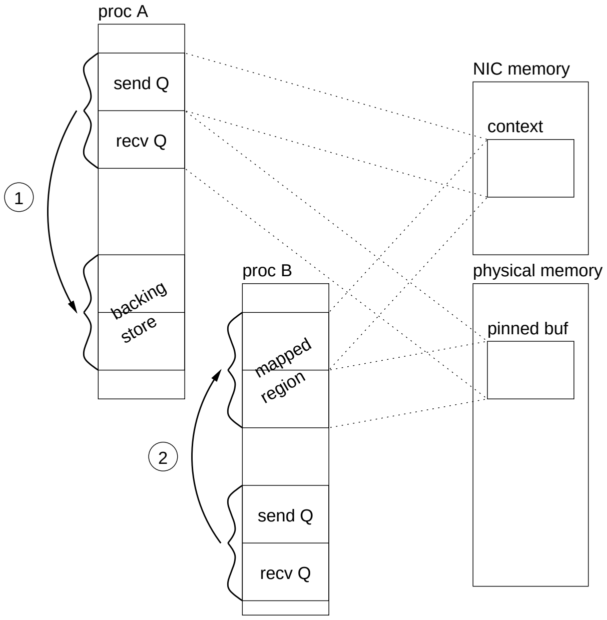

Figure 4. Copying buffers to switch from process A to process B.

The next stage is the call to COMM_context_switch to perform 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 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. However, we discovered that usually the queues are quite empty. This phenomenon can be explained by the fact that the send queue is filled by the host processor, which is also responsible for other processing, while it is emptied by the dedicated LANai processor. Therefore, the host processor cannot generate messages fast enough to fill the queue. The receive queue emptiness can be 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 and final stage of the context switch is the call to COMM_release_network. this 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 refilling 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. When all nodes are ready, the function returns and the new process is awoken with a SIGCONT.

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. Recall that the maximal number of credits is calculated by dividing the buffer allocated for the process between all the processes that can send it: $C_0 = \frac{B_r^{'}}{np}$, 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.

With gang scheduling the maximal number of processes that can send to a specific process is reduced to p (instead of np), so $C_0 = \frac{B_r^{'}}{p}$. 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.

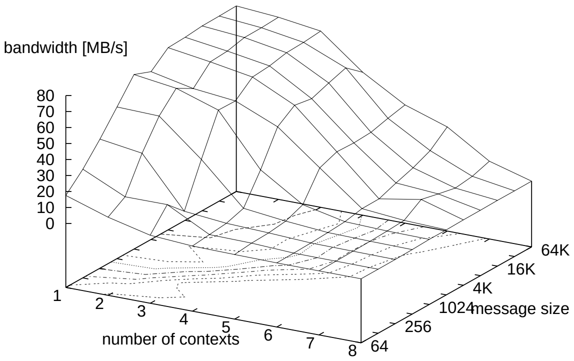

Figure 5. Bandwidth measurement for FM as a function of message size and the number of contexts, using the original FM buffer division.

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 5. 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 credit is used even if only part of each packet is used, so the system becomes

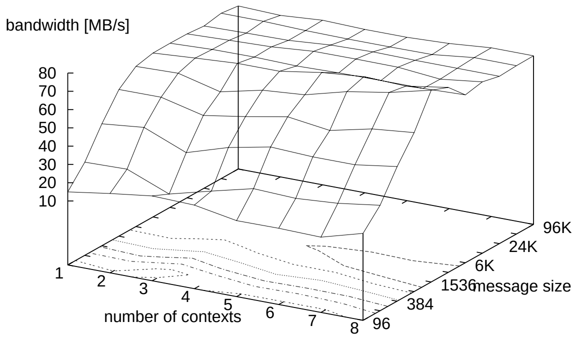

Figure 6. Total bandwidth as a function of message size and number of jobs, using the buffer switching scheme.

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.

We then measured the available point to point bandwidth when using our scheme. Recall that 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 6. This shows conclusively that when using the gang scheduling + buffer switch method, the presence of multiple applications 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.

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.

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 7, 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 refilling 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 led us to inspect if the copying the entire buffers is really necessary. Figure 8 shows that in fact 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 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 led us to improve our buffer switch algorithm: go through the buffers and only copy the valid pack-

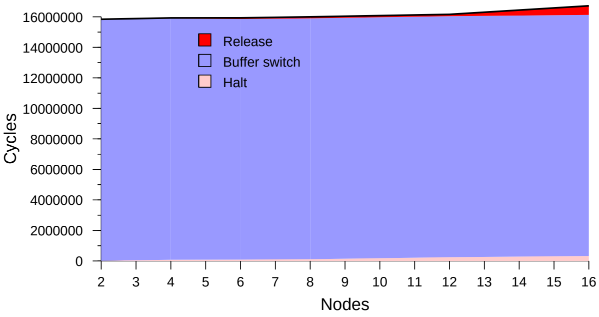

Figure 7. Time measurements for the buffer switch algorithm, in cycles.

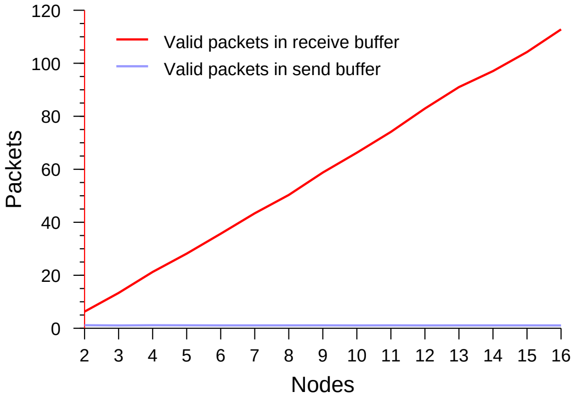

Figure 8. Number of valid packets in the buffers during buffer switching.

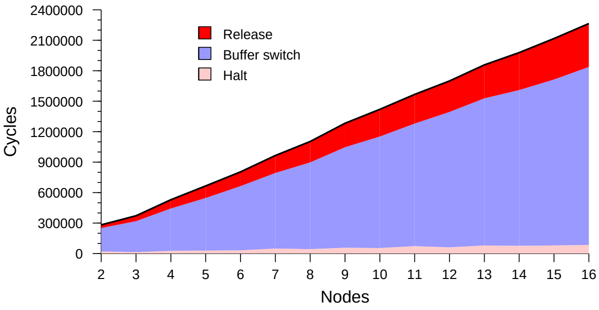

Figure 9. Time measurements for the improved buffer switch algorithm, in cycles.

ets. We measured the improved algorithm, whose results can be seen in figure 9. It can be seen that the time needed to switch the buffers is reduced dramatically, We also verified that the linear growth in the copying time is correlated with the linear growth of the number of packets found in the buffer.

These overhead measurements indicate that the overhead incurred by the buffer switch is negligible compared to the long time quantum used in multiprogrammed gang scheduling machines (seconds or even minutes). This overhead was even further reduced when we searched the buffers and only copied the valid packets. 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 [12]. The idea here is that instead of using gang scheduling, processes will be co-scheduled on the different 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 [9]. 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 [7]. 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, and by the Ministry of Science Basic Infrastructure Fund Project 9762. We are grateful to Prof. Andrew Chien for providing a source license of FM which enabled this research.

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] A. Hori, H. Tezuka, and Y. Ishikawa, “*Overhead analysis of preemptive gang scheduling*”. In *Job Scheduling Strategies for Parallel Processing*, pp. 217–230, Springer Verlag, 1998. LNCS vol. 1459.

[8] A. Kavas, D. Er-El, and D. G. Feitelson, “*Using multicast to preload jobs on the ParPar cluster*”. *Parallel Comput.*, 2001.

[9] C. E. Leiserson et al., “*The network architecture of the Connection Machine CM-5*”. *J. Parallel & Distributed Comput.* **33(2)**, pp. 145–158, Mar 1996.

[10] 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.

[11] 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.

[12] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, “*Dynamic coscheduling on workstation clusters*”. In *Job Scheduling Strategies for Parallel Processing*, pp. 231–256, Springer Verlag, 1998. LNCS vol. 1459.