Scaling MPI to short-memory MPPs such as BG/L

M.Farreras, T.Cortes, J.Labarta
Universitat Politecnica de Catalunya(UPC)
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
mfarrera,toni,jesus@ac.upc.edu

G.Almasi
IBM T.J. Watson Research Center
Yorktown Heights NY
gheorghe@us.ibm.com

Abstract

Scalability to large number of processes is one of the weaknesses of current MPI implementations. Standard implementations are able to scale to hundreds of nodes, but not beyond. The main problem in these implementations is that they assume some resources (for both data and control-data) will always be available to receive/process unexpected messages. As we will show, this is not always true, especially in short-memory machines like the BG/L that has 64K nodes but each node only has 512Mbytes of memory.

The objective of this paper is to present one algorithm that improves the robustness of MPI implementations for short-memory MPPs, taking care of data and control-data reception, the system will scale up to any number of nodes . The proposed solution achieves this goal without any observable overhead when there are no memory problems. Furthermore, in the worst case, when memory resources are extremely scarce, the overhead will never double the execution time (and we should never forget that in this extreme situation, traditional MPI implementations would fail to execute).

1. Introduction

The MPI (Message Passing Interface) [26] specification is a highly popular programming model for today's parallel machines ranging from commodity clusters to expensive supercomputers. MPI applications are divided into different tasks that communicate by sending and receiving messages among them.

Many implementations have been developed [16] [18] [14] [15] [9] [21] [17] [31], but memory management has been designed for hundreds of processors, and assumptions about resource availability are made that are no longer true for thousands of nodes.

ICS06 ,June 28-30, Cairns, Queensland, Australia.

In MPI programing model, the receiver process explicitly demands to receive a message (posting i.e. a MPI_Recv call). The point is that arriving messages need to match with posted receives. Therefore, when a new message arrives before the receive call has been posted (here and after unexpected message), some information needs to be stored somewhere to allow the matching (here and after control-data). Once the matching has been done, the space for control-data can be freed, but if a process receives lots of unexpected messages, even if only control-data is kept, the memory needed increases with the number of messages, which at its turn increases with the number of nodes.

Most current MPI implementations, assume to have enough memory to allocate control-data dynamically. However current MPPs trend to have short memory per node and large number of nodes, which makes memory a limitation, and control-data a problem that most current MPI implementations do not face.

A different problem comes together with control-data problem: When a message is unexpected, the message itself, the user's data, may be stored somewhere. This messages may also flood receiver's memory.

User's data problem has been faced before, although not always has a scalable solution been provided.

Finally, to test a real system where MPI has the above-mentioned scalability problems, let us use BGL supercomputer [30]. It provides a simple, flat, fixed-size 512Mb address space, with no paging, which limits the memory resources. This limitation requires a better control of the memory management, not only for data but also for control-data.

Although our work has been focused on BGL machine, page fault management is very expensive in terms of performance and supercomputers tend to disable the virtual memory mechanism, [7] [25] [5], therefore our solution may be applied to future designs. Moreover, some of our proposed solutions may be reused in designs sharing similar issues in other large scale MPPs or architectures based on embedded processors.

The goal of this paper is to describe the mechanism we have used to solve the above-mentioned problems and thus make MPI scalable to thousands of nodes, by providing MPI with a memory management protocol that will control both data and control-data memory management, without any overhead.

2. Related Work

In MPI designs there is always a trade-off between performance

and scalability, and memory management is a key issue because it affects both.

This paper is focused on MPI memory management, dealing with memory shortage in two different scenarios: control-data and data.

For every message arrival, control-data needs to be saved in order to manage the message. When memory is short thousands of control-data messages can flood the receiver's memory. Another memory management issue to be aware of relies in the user's data: In most current implementations, preference is given to the performance and control messages are avoided, so a message may be sent without receiver's permission and it is not guaranteed that the receiver will have enough memory to keep this message. Next paragraphs analyze both aspects of some MPI implementations.

Some contributions have taken the approach of statically allocate memory space to guarantee message arrival, this is the case for AIX [9], it allocates space for user's data but not for control-data. Also Fast messages in its credit-based control flow algorithm [29] preallocates space for both data and control-data for every other process in the system. PortalsMPI[28] pre-allocates space for control-data, while data is dropped if no memory is available. We should note though, that memory pre-allocation has the drawback that it does not scale well, just imagine if we decide to allocate 10Kb buffer per process in a 65536 node machine like BGL, each process would need to allocate 650Mb of memory, while BGL has 512Mb of available memory per node. If we reduce the memory allocated, and use it only for control-data, it is always a limitation that turns to be too restrictive in practice, especially as applications scale beyond a thousand processes.

The next logical step is to allocate the memory dynamically. Portals 3.1 softens the problem with control-data reception by allocating memory for control-data dynamically and the number of outstanding unexpected messages becomes dependent on memory space rather than count, not having enough memory to allocate control-data is not considered, (probably because to our knowledge it has scaled up to 1792 nodes), but memory is short in MPPs, and it turns to the problem we face on this paper for BG/L.

Regarding dynamic memory allocation for user's data three main directions have been taken. An on-demand credit management was explored for use in Fast Messages making the previous credit-based control flow mechanism more scalable[4]. The idea is distribute available credits on on-demand basis so it does not depend on the number of processors and each process receives credit according to its communication needs. Another direction uses prediction to achieve the same goal, buffers are dynamically allocated according to the predicted communication pattern of the application, and effort was put to analyze those communication patterns concluding MPI patterns being highly predictable [8][11][1].

Finally, most MPI implementations running on supercomputers use a rendezvous-based protocol [10][19][20][27]. In this mechanism, also called 3-handshake protocol, the sender asks permission to send the message and waits for confirmation, which guarantees data message arrival. Although all three directions are very interesting, they are only partially valid to our scenario since they all assume that control-data is able to be received at the other end, hence many changes would have to be done to cover control messages resulting in a completely new solution.

Facing both, control-data and data memory issues, we should mention the work from Myricom [31] in both MPICH-GM and MPICH-MX, and a contribution from Ohio State University [23]. MPICH-GM holds up to 256 eager messages, and both data and control-data messages are dropped afterward, to be resent after a timeout. Basically, the problem moves from a memory issue to network congestion. In MPICH-MX, if memory is short, sender is stalled, messages are copied in a buffer and a different thread will send them. Memory shortage has to be detected in advance to be able to apply this solution, so a threshold is defined. About the contribution from Ohio, a memory management protocol ("control flow") with dynamic pre-allocation of buffers is implemented. However, the solution has some weaknesses: it is not scalable because buffers are never freed and keep increasing resulting in a waste of memory (the authors are aware of it). Moreover, as far as we understand the paper explanations, it seems that the solution may have problems concerning the reception order of messages, it may violate MPI semantics (see section 4.3). And finally, all three approaches are adding overhead even when there are no memory problems.

To sum up, memory management for unexpected user's data has been updated to the current trend in some implementations running in supercomputers. However in all proposed solutions, rather some space is allocated a priori, which may waste critical memory resources or may be too restrictive, or memory is allocated on demand during program execution, which may scarify the robustness and reliability of the system. Only MPICH-MX solves both problems (data and control-data), the advantage of our solution is that while a threshold is defined in MPICH-MX, our threshold is dynamic and it depends on the available memory and the application needs.

3. Framework

In order to clearly understand the proposed solutions, we must first see the complete framework where we work. First we will present an overview of the machine Blue Gene Light, that will show the most important characteristics for our proposal. Then a quick overview of the design of MPI over the BGL machine and finally some relevant comments related to scalability and memory management.

3.1 BGL

In November 2001 IBM announced a partnership with Lawrence Livermore National Laboratory to build the Blue Gene/L supercomputer [6], a 65,536-node machine designed around embedded Power-PC processors. Through the use of system-on-a-chip integration coupled with a highly scalable cellular architecture, Blue Gene/L delivers 360 Tflops of peak computing power. Blue Gene/L represents a new level of scalability for parallel systems.

3.1.1 Hardware overview

The low power characteristics of BG/L permit a very dense packaging as shown in Figure 1. Two chips share a compute card that also contains SDRAM-DDR memory. Each node carries 512 MBytes of DDR memory.

Sixteen compute cards can be plugged in a node board. A cabinet with two midplanes contains 32 node boards for a total of 2048 CPUs. The complete system has 64 cabinets and 16 TB of memory. In addition to the 64K compute nodes, BG/L contains a number of I/O nodes (1024 in the current design). Compute nodes and I/O nodes are physically identical. The compute nodes of BlueGene/L are organized into a partitionable 64 x 32 x 32

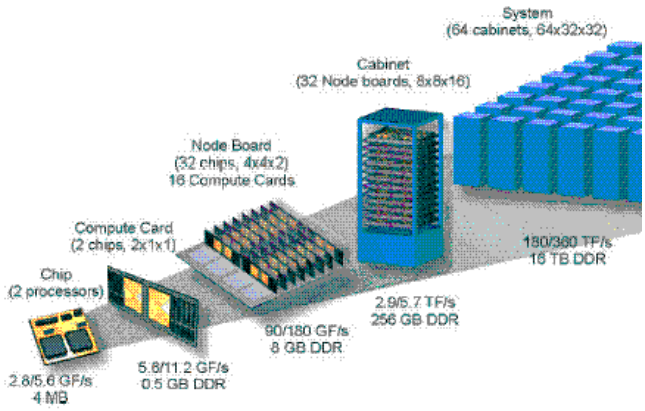

Figure 1. High-level organization of the Blue Gene/L supercomputer

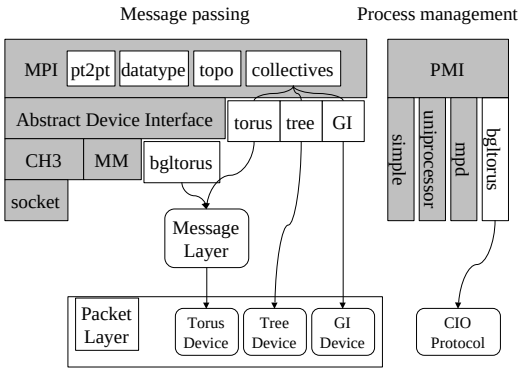

Figure 2. The BlueGene/L MPI architecture

three-dimensional torus network. Each compute node contains six bi-directional torus links for direct connection with nearest neighbors. The network hardware guarantees reliable and deadlock-free, but unordered, delivery of variable length (up to 512 bytes) packets, using a minimal adaptive routing algorithm. The I/O nodes are not connected to the torus network.

3.1.2 System software overview

The system software for Blue Gene/L [12] is a combination of standard and custom solutions. The I/O nodes execute a version of the Linux kernel for embedded processors and are the primary offload engine for most system services. No user code directly executes on the I/O nodes. Compute nodes execute a single user, single process minimalist custom kernel, and are dedicated to efficiently run user applications.

The control software running on compute nodes, it is a minimalist POSIX compliant compute node kernel (CNK) which provides a simple, flat, fixed-size address space with no paging. The kernel and application program share the same address space, with the kernel residing in protected memory. The kernel protects itself from the application by appropriately programming the PowerPC MMU.

3.2 MPI on BG/L

The Blue Gene/L communication software architecture [13] is divided into three layers. The torus/tree **packet layer** is a thin layer of software designed to abstract and simplify access to hardware by means of three main functions: initialization, packet send and packet receive.

The **message layer** is an active message system [33], built on top of the packet layer, that allows the transmission of arbitrary buffers among compute nodes.

The third component of the MPI software stack is **MPICH2**, an interface of MPI developed at Argonne National Laboratory. Figure 2 shows the software architecture of the MPI software stack.

A well known issue in the design of communication protocols is choosing at which level the different protocol functionalities are best implemented; in particular, the memory management mechanism has been the focus of considerable attention in previous projects (AM[33], U-Net [32], xKernel [22], [34], FM[29])

Because most of the needed information is in the abstract device implementation layer of the MPICH2 library (denoted `bgltorus` in Figure 2), the mechanism has been implemented in this layer, and only slight modifications have been done in the message layer.

3.2.1 Memory management: preliminaries

The design of the message layer was influenced by specific BlueGene/L hardware features, such as network reliability, packetization and alignment restrictions, out-of-order arrival of torus packets, and the existence of non-coherent processors in a chip. Together with these hardware features, there is the requirement for a low overhead scalable solution.

Scaling issues and virtual connections: In MPICH2, point to point communication is executed over virtual connections between pairs of nodes. A factor limiting scalability is the amount of memory needed by an MPI process to maintain state for each virtual connection. The current design of the message layer uses only about 50 bytes of data for every virtual connection for the torus coordinates of the peer, pointer sets for the send and receive queues, and state information. Even so, 65,536 virtual connections add up to 3 MBytes of main memory per node, not more than 1% of the available total (512 MBytes), just to maintain the connection table.

Communication protocol in the message layer: The protocol design is crucial because it is influenced by virtually every aspect of the BlueGene/L system: the reliability and out of order nature of the network, scalability issues, and latency and bandwidth requirements.

For every message, the transmitted information consists of a message envelope, containing control-data, and the data. The envelope is relatively small, and is delivered to the destination immediately. The data may or may not be delivered at once. As you can see in figure 3, BlueGene/L message layer offers three possibilities: eager, short or rendezvous protocols.

Because of the reliable nature of the network no acknowledgments are needed. A simple ”fire and forget” eager protocol is a viable proposition. Any packet out the send FIFO can be considered safely received by the other end. A special case of the eager protocol is represented by single-packet messages, which should be handled with a minimum of overhead to achieve good latency.

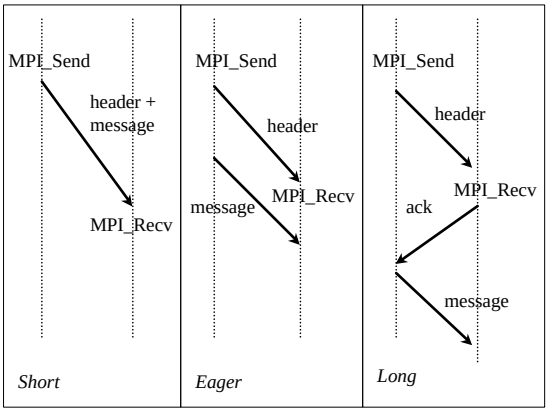

Figure 3. MPI Protocols.

The message protocol is also influenced by out-of-order arrival of packets. The first packet of any message contains information not repeated in other packets, such as the message length and MPI matching information. If this packet is delayed on the network, the receiver is unable to handle the subsequent packets, and has to allocate temporary buffers or discard the packets, with obvious undesirable consequences. This problem gets solved if the first packet is always acknowledged (i.e. in the rendezvous protocol) and next packets can be dynamically routed.

Memory usage issues: In the BlueGene/L MPI library no preallocation of memory is done, so unexpected messages, have to be received into a temporary buffer (called the *unexpected message buffer*) created dynamically. This is true for both eager and rendezvous protocols. Simultaneous unexpected messages from thousands of nodes can flood the receiver memory.

Moreover, there is another problem: even if the size of theses messages is 0 bytes, or the body of the message is not sent, they would consume some memory space because control-data, which is delivered immediately into the message envelope, needs to be saved until it matches a posted receive.

The outstanding control-data messages we need to support at any time increases with the number of processors, regardless of the size of the message, as memory is short, the control-data alone can flood the receiver memory.

The problem of simultaneous unexpected data messages has been faced before[29][4][24][3]. We need to solve this problem as well, the more suitable approach was selected to be implemented as a base for our memory management protocol. A credit-based flow control scheme was considered [4][24]. It would allow the receiver to control incoming traffic, overcoming eager protocol's main limitation and avoiding the 3-handshake protocol which usually increases latency. On the other side, an acknowledgment-based scheme of control flow [27][10] is more simple to implement with dynamic routing of data packets since it does not have to deal with the out-of-order arrival of packets. Therefore, an acknowledgment-based scheme of flow control was chosen for our memory management mechanism.

4. The proposed memory management algorithm

At this point we have shown that the performance and scalability of an MPI implementation may depend on the memory management techniques used by the library. Naive implementations of the MPI standard require more and more memory as the number of processors increases.

Our approach allows the MPI library to reside in a fixed amount of memory. We use a variant of the rendezvous protocol to implement memory control. The implementation must not degrade performance (e.g. by increasing latency) in the common case when memory is plentiful.

We start out with a short description of a simple rendezvous protocol that we used as the base case for our optimized implementation, this will solve the problem of user's data, and **then modify it to control memory usage for control-data, which is the main contribution of our work**.

4.1 High level overview of algorithm

The algorithm relies on an *Request To Send*, or RTS, packet to ask for permission to send a data message. Upon the arrival of an RTS packet the receiver is faced with one of multiple situations. A naive implementation of MPI reacts as follows:

- If a receive call matching control-data carried by the RTS already exists in the *posted request queue*, no extra memory is needed because the message will be received into the user buffer of the posted request. The receiver replies with a *Clear To Send* packet (*CTS*), permitting the sender to proceed with data packets.
- If the matching receive call has not been posted, but the receiver has plenty of memory, an MPI request and a message buffer are allocated in the *unexpected request queue*, to be matched against a future posting. A CTS packet is sent to the sender. Data is accumulated in the unexpected buffer.
- If the matching receive has not been posted and the receiver is short on memory, MPI aborts execution and prints a message about insufficient memory.

The matching algorithm described above is insufficient because (a) it uses heap memory for unexpected messages and (b) aborts as soon as it runs out of the heap.

Our optimized algorithm leaves the system's behavior unchanged in situations where memory is plentiful, but deals with low memory situations differently. Our aim is for MPI to make progress in these situations, avoiding deadlock or livelock while respecting MPI ordering semantics. We make use of the fact that MPI ordering semantics requires message *matching* to be in the order in which messages were sent but does not require message *delivery* to be in the same order.

Not enough memory: The first case we consider is when the matching receive has not been posted at the receiver and there is not enough memory to allocate an unexpected buffer, but there is enough memory to allocate an MPI request in the unexpected message queue. In other words, memory at the receiver is enough to perform the message matching but not enough to hold the actual message. The receiver does not allocate buffer memory and puts message delivery from the sender on hold.

The sender has to hold on to send data until a CTS packet arrives from the receiver. However, since message matching has been performed normally and in order, the sender is free to attempt to send subsequent messages to the receiver. Even though these later messages may actually complete sooner than the original refused message, this is not a violation of MPI semantics – after all message *matching* has been performed in order.

The message put on hold by the receiver will be delivered when the matching MPI receive is finally posted (which in a correct MPI program is bound to happen before the program ends).

Very little memory: The second case is more complex. Here is the novelty of our work. It occurs when the incoming message finds no matching receive, and the receiver is so short on memory that it cannot post the MPI request in the unexpected queue.

In this situation the receiver cannot perform the matching. It sends a *No Clear To Send Packet (NCTS)* and drops the message. The sender saves the message order by holding the message in the *pending send queue*. The sender is now responsible for keeping the order of the messages it is trying to send, and to retry them in order when prompted by the receiver. In effect we are extending the receiver's *unexpected* queue to the senders.

4.2 Data structures

Before we proceed to describe the algorithm implementation we need to introduce the data structures in our algorithm. The well known MPI message matching algorithm provides two request queues, one for messages *posted* by the receiver that have not yet arrived, and another queue for messages that were *unexpected* at the receiver but have arrived anyway.

Our algorithm is designed to cope with low memory situations at the receiving end. The intuitive idea is to extend the *posted* and *unexpected* queues to the senders in order to relieve the receiver from the burden of maintaining them in a low memory situation.

Thus we introduce an ordered *pending send queue* at every sender. Any MPI messages intended to be sent is first inserted into the pending send queue and is only removed when accepted into the receiver's own *posted* or *unexpected* queue.

The *failed match bit array* is an $O(n)$ array of bits (where n is the number of nodes in the MPI job). The bit corresponding to a process is set if message matching failed due to memory shortage at the destination.

The *age* array is another $O(n)$ array that is used to count the number of pending send queue restarts that have happened so far to a particular process. Every time the receiver recovers from a very low memory situation, communication is restarted with a different age to disambiguate message ordering.

Emergency memory is used by the receiver to communicate to the senders of pending messages when memory is low $O(1)$.

4.3 Detailed description

In this section we describe the algorithm in detail by following the path of a point-to-point message in our MPI protocol.

The message originates as an MPI request object at the sender, where it is enqueued into the pending send queue. Negotiation starts when an RTS (Request To Send) packet is sent from the sender to the receiver. The RTS packet carries control-data with the current age value of the sender. Upon arrival to the receiver a match against the *posted* queue is attempted. Multiple possibilities arise depending on the result of the match, available memory at the receiver and age discrepancy between sender and receiver.

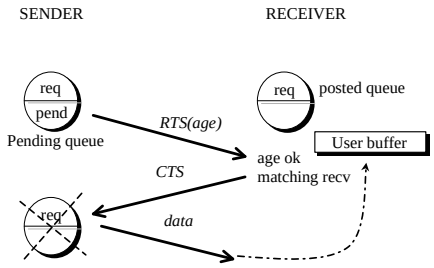

Figure 4. Age OK and matching MPI_recv

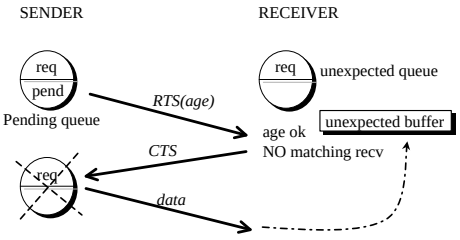

Figure 5. Age OK, no matching MPI_recv and enough memory

Successful message match: Figures 4 and 5 describe the relatively simple case when the incoming message at the receiver is either matched in the posted queue or else there is enough memory to hold the incoming message until it is posted (in which case it gets inserted into the unexpected queue). The age value carried by the RTSpacket must match the receiver's current age. Both these situations cause the receiver to reply with a CTS (Clear To Send) packet carrying the address of the (found or allocated) message buffer.

At the sender's site the MPI Request object is removed from the top of the pending send queue. This marks the end of the negotiation phase for this message. The sender proceeds to send the message data to the receiver in the usual way.

Age mismatch: Age mismatch means that memory conditions have changed since the message was sent, and so the message may be received out of order because previous messages may have been rejected. The protocol deals with this situation as follows: if the age value carried by the RTS packet does not match (figure 6), the receiver replies with a NCTS (Not Clear To Send) packet. At the sender this will cause the send request to stay in the pending send queue.

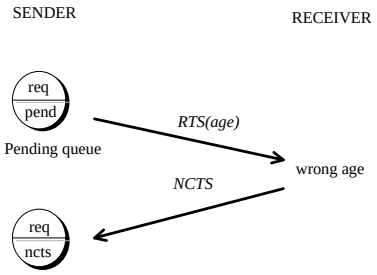

Figure 6. Age NOT OK and no matching MPI_recv

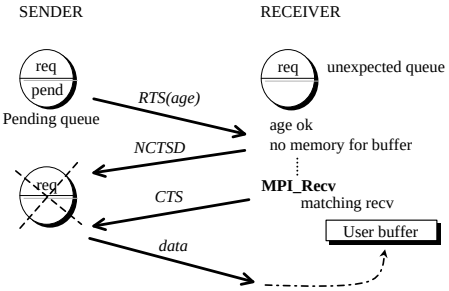

Figure 8. Enough memory for request but NOT for the buffer: delay data transmission

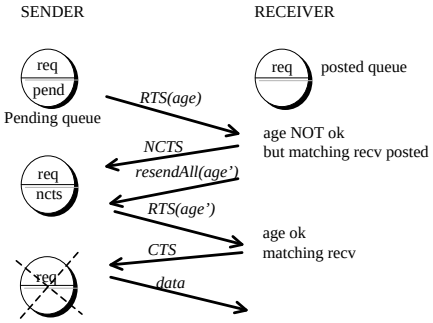

Figure 7. Age NOT OK and matching MPI_recv

However, even in case of an age mismatch the receiver still checks the `RTS` packet against the posted queue. If there is a match (Figure 7), the receiver sends off a `resendAll` packet as well. The sender copies the age value from the packet and re-sends all the messages currently in the pending send queue, in the correct order, with the new age value.

Receiver low on memory: Figure 8 shows the situation where the age matches and the receiver has enough memory to create a matching record in the unexpected queue, but not enough memory to hold the incoming message buffer. This causes the receiver to create an entry in the unexpected queue, using some memory from the heap. A `NCTSD` (Not Clear To Send Data) reply packet is sent back to the sender.

Upon receipt of a `NCTSD` packet the sender is allowed to remove the send request from the pending send queue (reader should notice that it differs from `NCTS` on this point). The sender is not allowed to send the actual data because there is no assigned buffer for it at the other side. However, since a record of the attempted match now exists at the receiver the sender is free to send other MPI send requests to the receiver.

Receiver very low on memory: Figure 9 displays the case where the receiver does not have enough memory to allocate a record of the missed match in the unexpected queue. The sender has queued the request into the pending queue before sending off the `RTS`. The receiver has to resort to *emergency memory* to send a `NCTS` reply. The receiver also sets the *match failed* bit for the sender.

The `NCTS` packet causes the sender to keep the current message in the pending send queue. This is a remedial action for making a record in the receiver's unexpected queue (the local pending send queue becomes an extension of the receiver's unexpected queue).

To allow the pending send queue to drain, one of two things has to happen at the receiver: either a new MPI request arrives into the *posted* queue or heap memory is freed through the `free()` library call (Figure 10). When either of these events happens, the senders have to be notified (by means of a `resendAll` packet) to re-try their message queues. The `resendAll` packet also carries a **new** age value to the senders. This ensures that stale data from the senders carrying the old age value will not be matched by the receiver, preserving MPI ordering semantics.

In order to ensure that the pending send queues are restarted when memory frees up in the receiver, our implementation intercepts calls to the standard `free()` library function, age is updated and `resendAll` message is sent.

Restarting the senders' pending send queues may require further use of the emergency memory by the receiver; in the situation where memory is very low but an MPI_Recv request is posted at the receiver communication has to restart without any additional memory resources. In this situation the `resendAll` packet is sent from emergency memory.

Emergency memory is not needed when communication is restarted because memory has been freed. If not enough memory has been freed to allow the sending of a `resendAll` packet it is likely not worth restarting communication.

Reacting to a `resendAll`: The sender starts sending all messages in the pending send queue whenever a `resendAll` packet arrives. The sender has to stop sending `RTS` packets when it reaches a send request that has not received an answer from the receiver yet. Valid answers are `CTS`, `NCTS` and `NCTSD`. If `CTS` or `NCTS` is received the message is de-queued and thus it will not be resent. The sender is forbidden from sending an `RTS` packet for any message in the queue that has not been replied to yet, because that would potentially violate MPI ordering semantics.

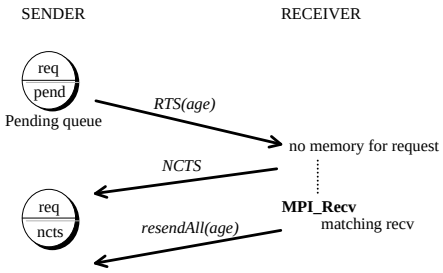

Figure 9. Not enough memory for request on receiver's side

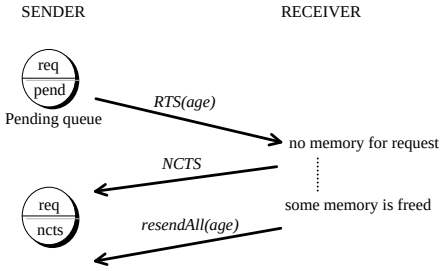

Figure 10. Some memory is freed

4.4 Deadlock issues

In this section we discuss how our improved algorithm differs from naive MPI message matching in terms of hanging behavior and deadlocks. Progress in an MPI program stops when the sender attempts to send a message but the receiver ignores or drops the data. This can happen in a number of situations.

- On BlueGene/L, messages *cannot* be lost during network transport because the network is reliable and guarantees the arrival of each packet.
- A situation in which a node stops draining the network *can* occur if an **incorrect** MPI program stops calling MPI routines. A BlueGene node will never stop draining incoming network packets as long as MPI routines are called. Thus, in a correct MPI program deadlock due to nodes sending data but not processing incoming packets cannot occur.
- **Incorrect** MPI programs can cause communication to be stalled permanently by not posting a matching receive for each sent message. Our algorithm ensures that deadlock does not occur if the program we are executing is correct. Although communication can temporarily stall because of lack of resources, a matching receive for *every* send will eventually be posted, causing communication to be restarted and the pending send queues to be systematically emptied. Therefore, deadlock due to insufficient memory resources in an MPI process cannot occur in a correct program. As long as receives are posted sender/receive stalls will be dissolved and communication will get restarted.

  We want to point that according to MPI specification [26] a correct MPI application should not rely in the underlaying implementation for its correctness, therefore using a acknowledge-based protocol for sending all messages does not change MPI semantics.

  However, we have to note that while **incorrect** MPI programs would cause the regular MPI implementation to overflow at the receiver and to abort execution, the advanced message matching algorithm may cause MPI to hang. We believe that this slightly increased risk of deadlock is worth taking in order to further scalability.
- *Livelock* (i.e. churning without making progress) is impossible in our optimized algorithm for similar reasons. In a situation where message matching fails due to lack of memory, and the receiver subsequently posts a new Receive request, the algorithm ensures that the pending sends from each sender will attempt to match the just posted receive. This is enough to ensure progress, because in a correct program the matching receive request *will* be posted.

5. Evaluation

The evaluation of the memory management protocol, will be done in two steps. First, we want to evaluate, how the mechanism works in an extreme situation, meaning an application that would not finish properly (crash, error) without the memory management mechanism. In order to do this testing we implement a microbenchmark.

Afterwards the mechanism will be tested under normal conditions. The NAS benchmarks will be used on this step because they do not show memory problems and we could measure the overhead added in the situation where the memory management protocol is not needed.

The NAS benchmarks are not usable to test the extreme case, because they are well balanced, meaning that the communication pattern is symmetric, so if a process is not able to receive it will not be able to send neither.

5.1 The killer microbenchmark

We will use a microbenchmark that recreates memory shortage conditions to prove the reliability of our mechanism and its behavior under this conditions.

The application used, figure 11, runs with n processors and all processors send a number of messages m, to the process number 0, this process is receiving the messages in the reverse order they have been sent. As a result, most of the messages either have to be received as unexpected messages or they have to wait for the matching MPI_recv to be posted.

Memory shortage condition is recreated by reducing the memory available. Figure 12, shows the overhead for different amounts of available memory (250Kb, which is the minimum amount of memory the application needs to run, 300Kb, 22Mb, 74Mb), the

```
if rank= 0
  for i = 1,N
      for j =1,number of processes
          MPI_Recv(from=j,tag=N-i)
else
  for i = 1,N
      MPI_iSend(to=0, tag=i)
  for i = 1,N
      MPI_Wait()
```

Figure 11. Killer benchmark structure

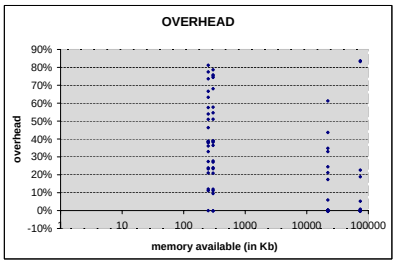

Figure 12. Overhead when memory management protocol is activated

situation without memory problems, with all memory available, was taken as a baseline. The application execution time was measured for different message sizes (1024, 10240, 102400), different number of messages (3 and 5) and different number of processors (16, 32, 64 up to 128).

We obtained a mesh of points and we can see that different overheads are not really related to the amount of memory available. Actually, once the memory management algorithm gets activated, factors such as restarts per message, message size, unexpected messages, etc affect the performance significantly and even in cases with very little memory available, a wide range of different overhead values is observed.

Nevertheless, the main conclusion is that the overhead is never more than twice the execution time without memory problems, which is not a hight price to pay to make your application run without problems.

We would like to point out that our killer microbenchmark does not work neither in the standard MPI implementation running on BG/L nor in MPICH-MX. The implementation on BG/L and its problems have been deeply discused. About MPICH-MX, sender process is stoped when a memory problem occurs, hopping that receiver will free some memory but it may happen that, like in our benchmark, memory is not freed until it resolves the first receive and some memory needs to be allocated in the receiver process for that. A deadlock situation is reached, therefore the application hangs.

Moreover, this result is specially significant because the application is message bound and no computation is done. If it had more computation, the effect of the message overhead would be even smaller.

5.2 NAS benchmarks

In order to evaluate the overhead in applications without memory problems, the NAS-MPI benchmark suite was used. We use class B problem size of the NAS Bt, Cg, Ft, Ep, Mg, Lu, and Sp benchmarks [2]. Figure 13 shows the execution of each benchmark, with 16, 32, 64 and 128 processors. First column is the default MPICH solution, the messages are sent using different protocols (short, eager or rendezvous) depending on the message size. The second column shows the execution of each benchmark sending all messages using the rendezvous protocol regardless of the message size. And the third execution is our memory management protocol, where all messages are also sent using the rendezvous protocol but with the memory management mechanism.

As it is shown in figure 13, most of the times there is no overhead, this is the case for Bt, Ep and Sp benchmarks, where the proposed solution is as performant as the default solution, we also notice that for these benchmarks the execution sending all messages using the rendezvous protocol has good performance as well. Regarding the rest of the benchmarks, Cg, Ft, Lu and Mg, figure 13, we notice an overhead of 1% in the worst case, we realize that most of the overhead is due to the use of the rendezvous protocol. And another good news is that this slight overhead is decreasing with the number of processes, we observe it in Ft, and Mg benchmarks.

6. Conclusions

We have shown the necessity of controlling the correct arrival of any message, containing either data or control-data, for current trend MPP machines like BG/L.

We have proposed a memory management protocol for BG/L overwhelming the limitation of short-memory MPPs in order to gain scalability. Our solution allows MPI to support as outstanding unexpected messages as memory is available not only in the receiver node but also in all senders. And we have shown that this mechanism works, without any overhead under normal conditions (no memory problems) and within less than twice the execution time, for an application that would crash without the proposed memory management protocol.

7. Future Work

In this paper we have focused, for the sake of clarity, on the problems that appear due to memory shortage and how to react when the system reaches a state were no memory is available. We have not tackled issues such as network saturation that could also produce problems in the advance of the application. Although they could be handled in a improved version of our algorithm, we preferred to leave it as separate work because it is not a problem of memory shortage. Network saturation can appear even when memory is available. Nevertheless, we plan to upgrade our algorithm to handle this situation in the future, although we have not detected the problem in any of our experiments.

Acknowledgments

We thank Xavier Martorell for the helpful discussions and technical support.

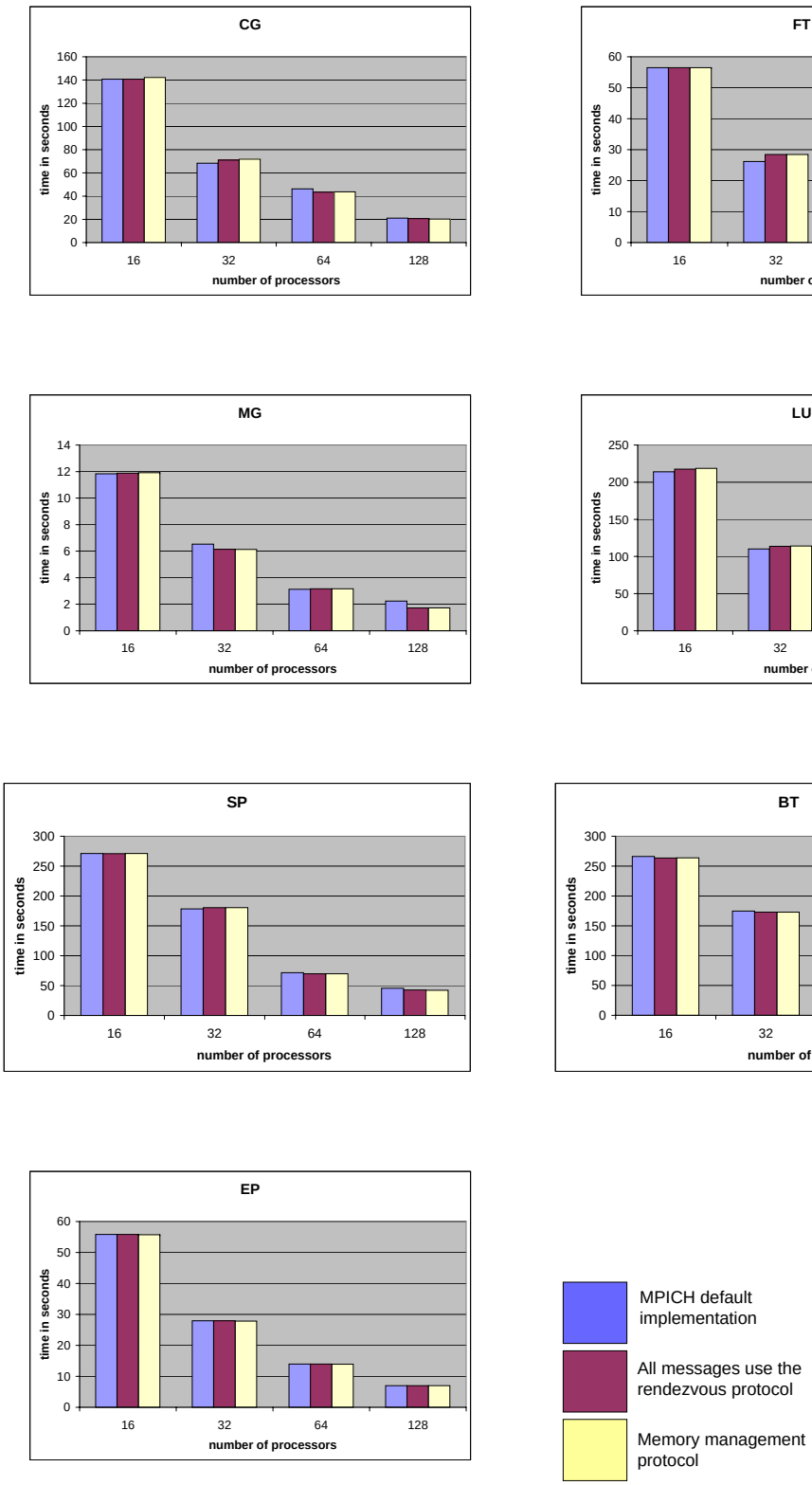

Figure 13. Time in seconds of the class B of NAS benchmarks Cg Ft Mg Lu Sp Bt and Ep, comparing the standard solution, the standard sending all messages with the rendezvous protocol and the solution with the memory management protocol

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01

8. References

[1] A. Afsahi and N. J. Dimopoulos. Efficient communication using message prediction for cluster multiprocessors. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 162–178, 2000.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM Press.

[3] R. Brightwell, R. Riesen, and A. B. Maccabe. Design, implementation, and performance of MPI on Portals 3.0. *International Journal of High Performance Computing Applications*, 17(1):7–20, 2003.

[4] R. Canonico, R. Cristaldi, and G. Iannello. A scalable flow control algorithm for the fast messages communication library. In *CANPC '99: Proceedings of the Third International Workshop on Network-Based Parallel Computing*, pages 77–90, London, UK, 1999. Springer-Verlag.

[5] Cray. Red storm: Catamount lightweight kernel.

[6] N. R. A. et al. An overview of the blue gene/l supercomputer. In *Proceedings of Supercomputing 2002*.

[7] R. et al. Cougar, a lightweight operating system running on asci red.

[8] Felix Freitag, Jordi Caubet, Montse Farreras, Toni Cortes, Jesus Labarta. Exploring the predictability of mpi messages. *IPDPS '03: Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[9] I. M. P. E. for AIX. MPI Programming and S. Reference. Ibm mpi programming guide.

[10] M. for the Earth Simulator Home Page. http://www.ccrl-nece.de/ ritzdorf/mpies.shtml. Mpi/es hp.

[11] F. Freitag, J. Corbalan, and J. Labarta. A dynamic periodicity detector: Application to speedup computation. In *IPDPS '01: Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 2–2.

[12] R. B. e. a. G. Almasi. An overview of the blue gene/l system software organization. In *Proceedings of Euro-Par 2003*, 2003.

[13] J. C. e. a. George Almasi, C Archer. Implementing mpi on the bluegene/l supercomputer. In *Proceedings of Euro-Par 2004*.

[14] L. home page. http://www.lam mpi.org. Lam.

[15] O. M. home page. http://www.open mpi.org. Lam.

[16] M. home page. http://www.unix.mcs.anl.gov/mpi/mpich. Mpich.

[17] C.-M. H. P. http://www.cray.com/. Cray hp.

[18] C. P. http://www.epcc.ed.ac.uk/epcc projects/CHIMP. Chimp.

[19] A. P. http://www.llnl.gov/asci/platforms/white/. Asci white home page.

[20] A. P. http://www.sandia.gov/ASCI/Red/. Asci red home page.

[21] S. M. P. T. M. http://www.sgi.com/products/software/mpt/. Sgi hp.

[22] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[23] D. P. Jiuxing Liu. Implementing efficient and scalable flow control schemes in mpi over infiniband.

[24] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for atm networks: credit update protocol, adaptive credit allocation and statistical multiplexing. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 101–114, New York, NY, USA, 1994. ACM Press.

[25] S. N. Labs. Performance comparision between sunmos and traditional linux kernel.

[26] J. h.-f. MPI: A message Passing Interface Standard. Message passing interface forum.

[27] D. of Computer Science and E. T. O. S. U. http://nowlab.cis.ohio-state.edu/projects/mpi iba/. High performance mpi on iba mpi over infiniband project.

[28] M. on CPlant. http://www.cs.sandia.gov/cplant/doc/mpich portals3.html. Mpich 1.2.0 over portals 3.1.

[29] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 55.1, 1995.

[30] I. research. http://www.research.ibm.com/bluegene/. Ibm research blue gene project page.

[31] M. Software and D. H. P. M. http://www.myri.com/scs/. Myricom hp.

[32] A. B. e. a. T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *1995: Proceedings of the 15th ACM SIGPLAN symposium of Operating Systems Principles*, 1995.

[33] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[34] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical report, Ithaca, NY, USA, 1997.