

Exchanging multiple messages via MPI

Joachim Hein, Stephen Booth, Mark Bull

EPCC

The University of Edinburgh

Mayfield Rd

Edinburgh EH9 3JZ

Scotland, UK

October 13, 2003

Abstract

In a series of benchmark codes we demonstrate that sending multiple messages is most efficiently done by using non-blocking communication. It is irrelevant whether these messages go to the same processing node or to different ones. The use of non-blocking communication allows the latencies of the individual communication calls to be overlapped and better utilises the dual plane design of the HPCx switch network. These findings can be applied to, for example halo exchanges in domain decomposition codes. We demonstrate that additional modest performance gains may be achieved by the overlap of communication and calculation.

1 Introduction

High performance scientific software has to use the parallel architecture of today's super computers to achieve the required capability levels. This requires the communication overheads to be extremely small, if the code is to operate efficiently on hundreds of processors. Therefore it is of great interest to have fast means to communicate the data between the processors.

In many scientific codes one encounters the situation that a given processor has to send data to several other processors. The halo exchange in a domain decomposed parallel code is an example of such a situation. In this paper we report on a detailed investigation of how to send multiple messages from a single processing node efficiently on the HPCx hardware.

We start this report with an overview of the relevant hardware of the HPCx system. In the following chapters we use a series of benchmark codes, to investigate which MPI calls perform better than others in such a situation and the reasons why they perform better. We will demonstrate that our results lead to a significant improvement in a "*closer to life*" example of a Jacobi inverter for a domain decomposed matrix in 2 dimensions. We conclude with an investigation of

whether further improvements can be achieved by overlapping communication and calculation calls.

2 The HPCx system

The HPCx system consists of 40 IBM p690 Regatta H frames. Each frame has 32 POWER4 processors with a clock of 1.3 GHz. This provides a peak performance of 6.6 Tflop/s and up to 3.2 Tflop/s sustained performance. The processors inside the frames are grouped into 4 multi-chip-modules (MCM), with 8 processors each. To increase the communication bandwidth of the system, the frames have been divided into 4 logical partitions (LPAR) coinciding with the MCMs. Each LPAR is operated as an 8-way SMP, running its own copy of the operating system AIX.

The interconnect is a dual plane IBM SP Switch2 (colony) network. Each LPAR is connected to the network with two PCI switch adapters, one for each plane. Processors inside a logical partition can communicate via shared memory. Processors from different LPARs have to communicate via the switch network and all data has to be send through the switch adapter pair. This is substantially slower than using the shared memory. In this report we will focus on the performance of the message passing between processors from different LPARs via the switch network. For many applications which use a large number of processors the time spent on the network communications determines the overall performance.

According to the MPI standard, a standard send can be implemented as a buffered send or as a synchronous send. On the HPCx system a standard send is implemented as a buffered send for small messages and as a synchronous send for large messages. The threshold for the protocol switch can be controlled with the environment variable `MPI_EAGER_LIMIT`. The maximum possible value of `MPI_EAGER_LIMIT` is 65536. Using a buffered send (also known as an eager send) has the advantage of a three times lower latency than using a synchronous send. The disadvantage is that a large value of `MPI_EAGER_LIMIT` increases the memory consumed by the MPI library. The default values are very small and we strongly recommend tuning this value for you application. Please note that the maximum value of `MPI_EAGER_LIMIT` depends on the number of MPI tasks.

For this investigation we used version 5.1 of the operating system AIX and version 3.2 of Parallel Environment for AIX.

3 Multi-Ping-Pong

We start the investigation with a variation of the well know ping-pong benchmark, which we call “multi-ping-pong”. For this benchmark, we use two processors from different LPARs. The other processors of these two LPARs are kept idle. The first processor sends n messages to the second processor. Once all n messages have been received the second processor sends these n messages back to the first processor. We report on the times for a whole cycle averaged over several hundred cycles, ensuring warming up effects of the caches are negligible.

We used four different strategies to implement the sending part of the benchmark. The first implementation is using n calls to `MPI_Send` and the second implementation makes n calls to `MPI_Isend` followed by a single call to `MPI_Waitall` on all n outstanding MPI calls. For the third and fourth implementation we used `MPI_Ssend` and `MPI_Issend` respectively. Again

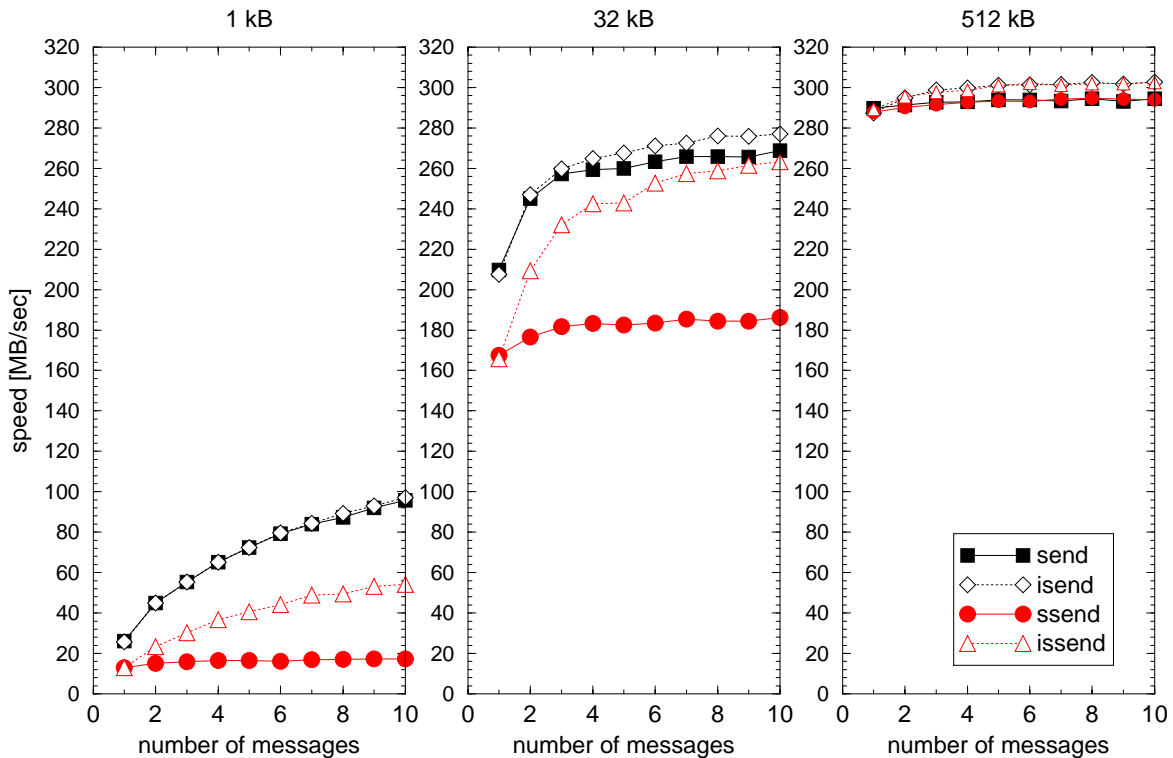


Figure 1: Speed of data transfer in the multi-ping-pong benchmark. The size of the individual messages is given above the graphs.

when using the non-blocking call we used `MPI_Waitall` to clear the communications. For the receiving part we used n calls to `MPI_Irecv` followed by a single call to `MPI_Waitall` in all cases.

This benchmark has been investigated for a wide range of message sizes. We show the results for three message sizes in Figure 1. The speed of the data transfer is calculated as

$$\text{speed} = \frac{2n \times \text{message_size}}{\text{time}(\text{cycle})}.$$

For this benchmark we set `MP_EAGER_LIMIT=65536`. Hence messages up to 64 kB are send using the eager protocol¹.

When using `MPI_Send`, `MPI_Isend` or `MPI_Issend` to send messages of a few kilo-bytes Figure 1 shows a substantial performance increase when sending several messages before starting to send messages back. These performance increases can be dramatic. Sending 10 messages of 1 kB with `MPI_Isend` before sending them back results in a 4 times faster communication compared to sending a single message back and forth. Below the `MP_EAGER_LIMIT` one observes `MPI_Send` and `MPI_Isend` perform superior to `MPI_Issend`, which is not surprising, given the more complex nature of non-blocking synchronous send when compared to a standard buffered send. For `MPI_Ssend` there is only a very small increase in performance when

¹Rerunning the benchmark with a different value for `MPI_EAGER_LIMIT` would not give new insights. The performance of the synchronous send does not depend on the value of `MPI_EAGER_LIMIT` and a standard send performs identically to the synchronous send above the `MPI_EAGER_LIMIT`.

sending several messages instead of only a single message. This marginal increase is either due to the improved efficiency of several calls to `MPI_Ssend` at the sending site or to having several non-blocking `MPI_Irecv` calls at the receiving site. Since the improvement is so small, we hold it is not worthwhile to investigate this further. For large message sizes the rate saturates at ≈ 300 MB/s, which is close to the hardware limit of a pair of switch adapters.

When using non-blocking communication, or `MPI_Send`, below the `MP_EAGER_LIMIT`, the sending MPI call returns almost immediately and the next sending call can be issued while the data of the former calls is still in transit. This results in an overlap of the subsequent MPI calls, as we will see in Section 4. On the other hand, when using `MPI_Ssend`, messages have to be send back and forth between sender and receiver, before the `MPI_Ssend` can return and the next `MPI_Ssend` can be issued. There is little potential for overlapping subsequent calls, which explains the poor performance increase when sending multiple messages with a synchronous send.

4 Switch planes and latencies

In this section we discuss further the reasons behind the improvement when using `MPI_Isend`. Figure 2 shows the time for a cycle when sending a fixed amount of data. Note that this is different from the previous section, where we kept the size of the messages constant. Two of the curves compare the performance of sending a single message back and forth using either one or two switch planes². The results show the second plane is only used once the size of the messages exceeds 8 kB. The curve for sending 1 message on 1 plane can be reasonably described with a latency-bandwidth model. The explanation of the minor bumps and dents in this curve goes beyond the scope of this report.

When using two planes to send a single message, the curve is extremely flat for messages between 8 kB and about 12 kB, with a second plateau for messages larger then 24 kB. The explanation must be, that the MPI system divides messages into pieces of 8 kB. Messages of below 8 kB are sent in one piece, which can utilise one of the planes only. Messages between 8 kB and below 16 kB get divided into two pieces, one of 8 kB and a second smaller piece. Each piece gets sent via one of the switch plane and the total time of the transfer is determined by the time it takes for the 8 kB piece to be sent. This results in the plateau discussed above. Once a message exceeds 16 kB, three pieces are needed: two pieces of 8 kB each and one for the rest. The two 8 kB pieces are sent via the two planes and the time it takes to send the remainder causes the slope between 16 kB and 24 kB. For messages larger than 24 kB we observe a second plateau due to having an odd number of 8 KB pieces and a smaller piece to send the remaining data.

Let us now turn to the remaining curve, sending two messages via both switch planes. In this case the individual message sizes are exactly half the size of what we used in the single message runs. The figure shows, for small messages the latency is essentially the same as for single messages. When splitting the data into two messages, the second switch plane gets utilised earlier, which leads to an improvement for this sending mode in the region between 4 kB and 10 kB.

Once the total data volume of the two messages exceeds 16 kB, we observe a substantial time increase. At this point the MPI library splits each of the two messages into more than one piece.

²For a production run, users should always use both switch planes.

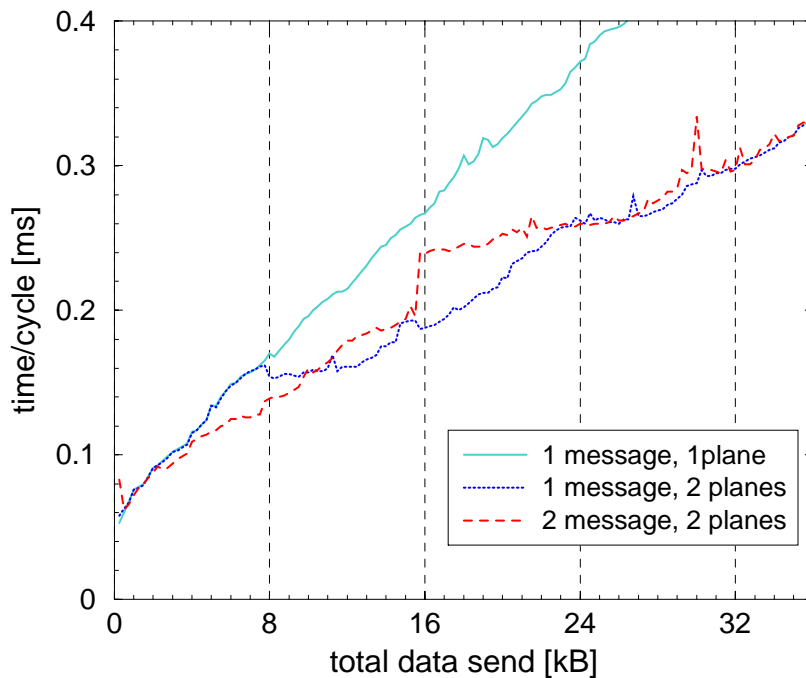


Figure 2: Effect of using one or two planes on the multi-ping-pong benchmark. The results are for using MPI_Isend and MPI_Waitall.

Due to the meta-data needed to make the MPI system work, this occurs slightly before 16 kB. It appears the larger 8 kB pieces of both messages use the same switch plane, which is clearly not optimal. This reasoning is supported by the curve being almost flat until 27 kB. At this point the performance of sending in one or two messages is about equal. This is understandable since this is the region of the second plateau for sending as a single message.

In this section we have established two effects which lead to an increased efficiency when issuing several non-blocking MPI calls before clearing the waits. Firstly it is possible to overlap the latencies of the calls, and secondly for small messages this results in a better utilisation of the dual plane switch architecture.

5 Splitting messages

Having established in sections 3 and 4 that it is faster to issue several non-blocking MPI calls than sending single messages back and forth, it is obvious to ask whether this improvement makes it worthwhile to split larger messages into several smaller messages. This is easy to investigate from the data we collected with the “multi-ping-pong” benchmark.

In Figure 3 we give the ratio of the bandwidth for sending a fixed amount of data in two messages or one message. The figure shows a number of features. Splitting into messages of less than 0.5 kB has a very bad impact on the performance.

For messages between 4 kB and 10 kB splitting up seems very favourable. As discussed above, this is due to the better utilisation of the dual plane architecture. It has to be pointed out, that this is a specific feature of the (multi-)ping-pong benchmark. In standard ping-pong there is

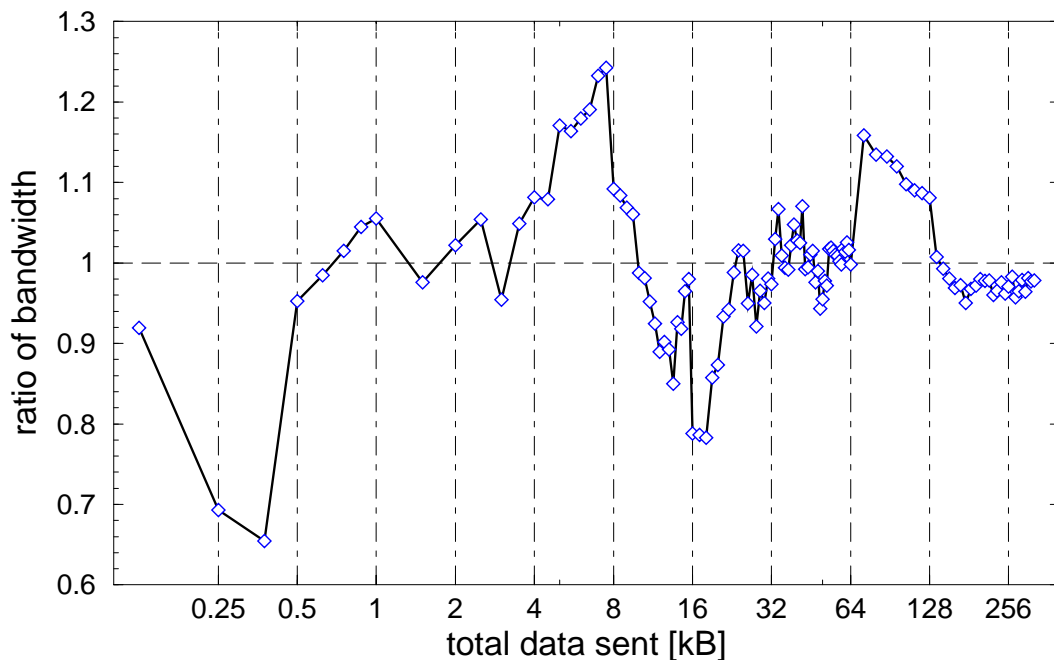


Figure 3: Ratio of the bandwidth for splitting the data into 2 message divided by the bandwidth for sending as a single message. The figure is for an `MP_EAGER_LIMIT=65536`.

only one message to be sent at any given time. If, however, your application has more than one message to send at a time per LPAR, which is all the 8 processors, this performance increase will not show, since the other messages will utilise the second switch plane.

The next feature we would like to point out is the marked drop for a data size between 16 kB and 24 kB. This drop is a direct consequence of the step in Figure 2 when sending as two messages and the reasons have been explained above.

The last feature we would like to discuss is the peak between 64 kB and 128 kB. By splitting such messages into two pieces, the individual pieces will send eagerly, resulting in a performance increase of more than 10%. Whether this improvement will show in a real application, which sends more than one message per LPAR at a given time is a question which goes beyond the scope of this report.

In Figure 4 we show the bandwidth when splitting a message into up to 8 smaller messages, while keeping the total data sent back and forth between sender and receiver fixed. Overall, the curves on the graph are quite flat. This indicates that by sending many small messages together in the fashion of this benchmark, a performance comparable to sending a single large message can be achieved. This is a consequence of the overlapping latencies when issuing several non-blocking `MPI_Isend` calls.

Returning to the question of whether splitting messages into pieces is advantageous, we want to discuss a number of details in Figure 4. Splitting a message of 1 kB into eight tiny pieces of 128 bytes each is clearly not beneficial. Also splitting larger messages into smaller ones of 8 kB has a negative impact on the performance, see the results for sending 16 kB in two pieces, 32 kB in four pieces and 64 kB in eight pieces. As discussed, this drop is a consequence of the way the MPI library splits messages into pieces of up to 8 kB and the way it distributes them onto the

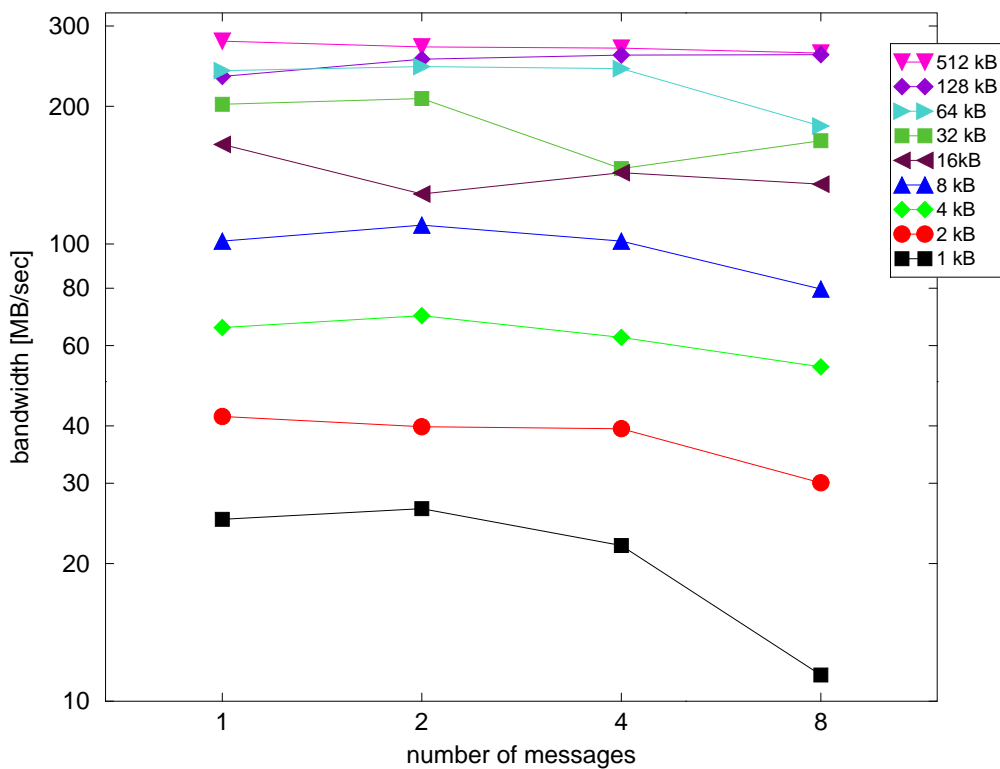


Figure 4: Performance of the “multi-ping-pong” benchmark when sending a fixed amount of data in a different number of messages. The different symbols specify the total amount of data send back and forth within one cycle. The results are for using MPI_Isend and MPI_Waitall.

two switch planes.

On the other hand, splitting a message of 128 kB into two pieces of 64 kB gives a marked improvement. A similar improvement has been observed for splitting a 256 kB message into 4 pieces of 64 kB. The latter has been omitted from the figure for reasons of clarity. As explained, by splitting the data into messages of maximally 64 kB, the individual messages are below the `MP_EAGER_LIMIT`. In this context it is interesting to note that for splitting a 512 kB message into 8 pieces of 64 kB no such improvement is observed.

Before moving onto the next benchmark code, we need to summarise our key results. With the multi-ping-pong benchmark we have shown that by using non-blocking communication it is possible to overlap the latencies of several MPI calls and to make better use of the dual plane architecture for messages below 8 kB. Using non-blocking communication for several small messages it is possible to achieve a performance level comparable to sending all the data in a single larger operation.

6 Sending around a ring

In the multi-ping-pong benchmark all messages sent from a processor go to the same receiving processor. In this section we address the case where the messages go to different receiving processors. To investigate we arrange a number of processor on a ring. Each processor sends a message to his left and his right neighbour. This has been implemented in five different ways:

Pattern 1: Sending to the left around the ring, wait to finish and sending right around the ring. Use non-blocking standard send:

- `MPI_Irecv` from right, `MPI_Isend` to left, `MPI_Waitall` on the two communications, then
- `MPI_Irecv` from left, `MPI_Isend` to right, `MPI_Waitall` on the two communications

Pattern 2: Basically the same idea as Pattern 1, but implemented with `MPI_Sendrecv`:

- `MPI_Sendrecv` sending to the left and receiving from the right, then
- `MPI_Sendrecv` sending to the right and receiving from the left

Pattern 3: Now we try to overlap the communications by using non-blocking sends and postpone the wait until all messages have been send:

- `MPI_Irecv` from right, `MPI_Irecv` from left, then
- `MPI_Isend` to left, `MPI_Isend` to right and finally
- `MPI_Waitall` on all four communications

Pattern 4: Similar to Pattern 3 but with non-blocking synchronous send `MPI_Issend` instead of non-blocking standard send `MPI_Isend`

Pattern 5: Similar to Pattern 3 but with non-blocking buffered send `MPI_Ibsend` instead of `MPI_Isend`

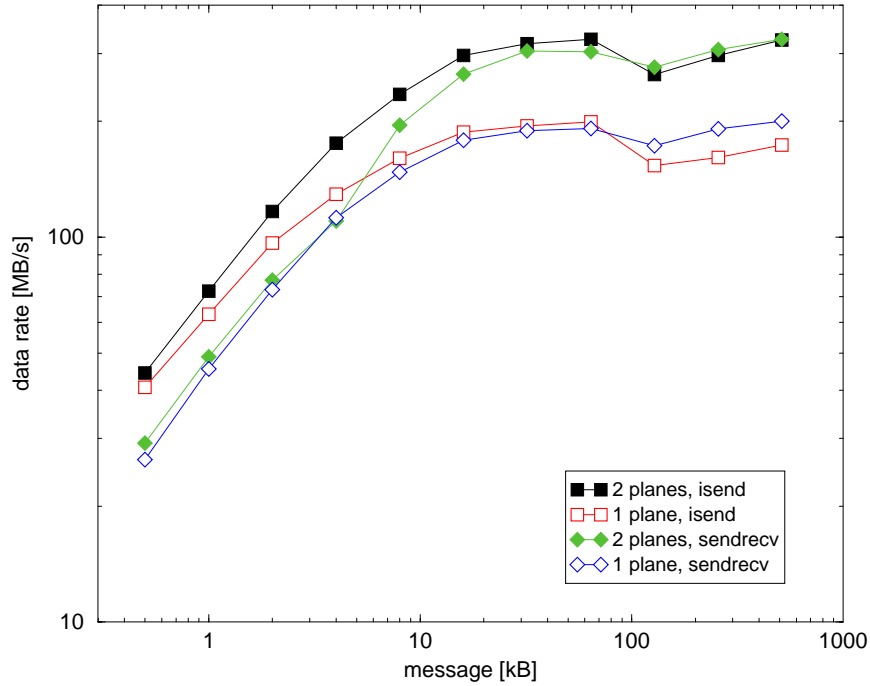


Figure 5: Comparing the performance of the ring bench mark for one and two active switch planes. Data are for 4 active processors from different LPAR and `MP_EAGER_LIMIT=65536`.

These communication patterns have been tested with eager and non-eager sending. For messages up to 64 kB we observed four distinct classes of performance

- i. Communication Patterns 3 and 5 perform similarly. For messages below the `MPI_EAGER_LIMIT` these patterns give the best performance.
- ii. The second best performance is observed for Patterns 1 and 2.
- iii. It is interesting to see that the performance of Pattern 4 does not depend on the value of `MPI_EAGER_LIMIT`. Its performance is similar to the non-eager version of Patterns 3 and 5.
- iv. The worst performance is observed for the non-eager use of Patterns 1 and 2.

Since there is no eager sending beyond 64 kB, for messages above this size, there are only two classes of performance, if one discounts communication Pattern 5. For messages larger than 64 kB the observed performance of `MPI_IbSend` is poor.

In Figure 5 we show details of the performance of Pattern 2 and 3 using eager sending up to 64 kB and non-eager sending beyond. These two curves represent the two best performing classes (i and ii). These results have been obtained using 4 processors from 4 different LPARs. The figure also displays the effect of using one or two switch planes. For small messages, the number of switch planes used has little effect on the performance. The different performance of the two patterns is caused by the overlapping of the latencies when using the non-blocking pattern. The figure shows further that when finishing one message at a time, which is what

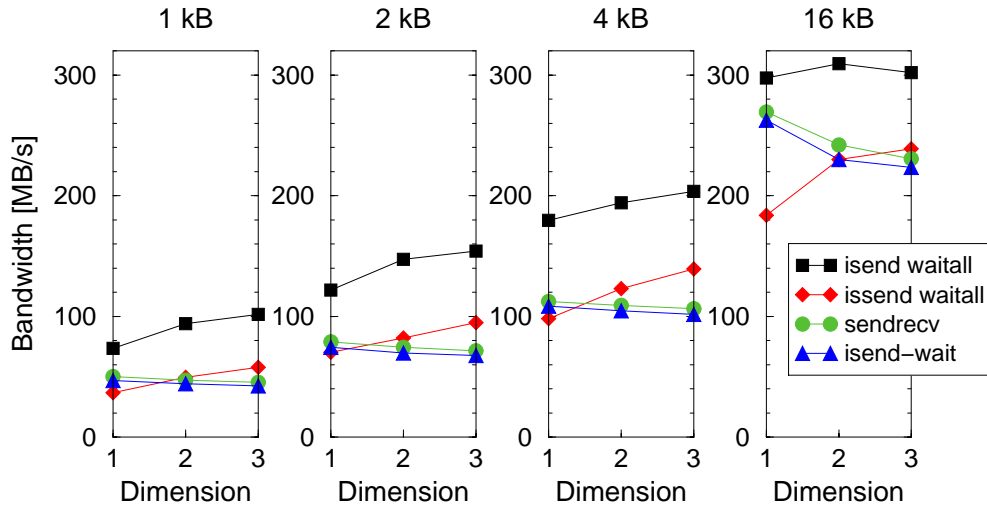


Figure 6: Sending data around a torus. The graphs display the bandwidth achieved for different numbers of dimensions. The size of the individual messages is given in top of each graph. The torus extends for 3 processors in each dimension. All processors are from different LPARs.

Pattern 2 does, the second switch plane only improves performance for messages larger than 8 kB. These are exactly the same findings as for the Multi-Ping-Pong bench mark.

The figure indicates that for large messages beyond the `MPI_EAGER_LIMIT` the Pattern 2 might perform slightly better than Pattern 3. This might be caused by the sorting needed on the receiving end to associate the individual packages arriving with the appropriate message buffer. Since the difference is small we did not investigate this further.

We tested the dependency of the performance of the five patterns on the size of the ring. Using 1 or 8 active tasks per LPAR, we checked over a range from 2 to 64 LPARs. We did not observe a significant dependency of the performance on the size of the ring.

To summarise this section, we have shown that our findings with the Multi-Ping-Pong bench-mark hold even when the messages go to different processors. We will investigate this further in the next section.

7 Sending around a torus

In this section we show that the findings of the previous chapter still hold when we arrange the processors on a torus in several dimensions. This has been investigated for four different communication patterns which are generalisations of the Patterns 1 to 4 of Section 6. We display our results for message sizes up to 16 kB in Figure 6. The results are averaged over a large number of cycles around the torus. For reasons which go beyond this investigation, we did not obtain a clear picture for messages sizes beyond 64 kB and have excluded these results from the figure.

When using non-blocking standard send or non-blocking synchronous send and a single call to `MPI_Waitall` for messages of a few kilo-bytes, the performance increases when increasing the number of dimensions of the torus. When using non-blocking `MPI_Isend` we reach a per-

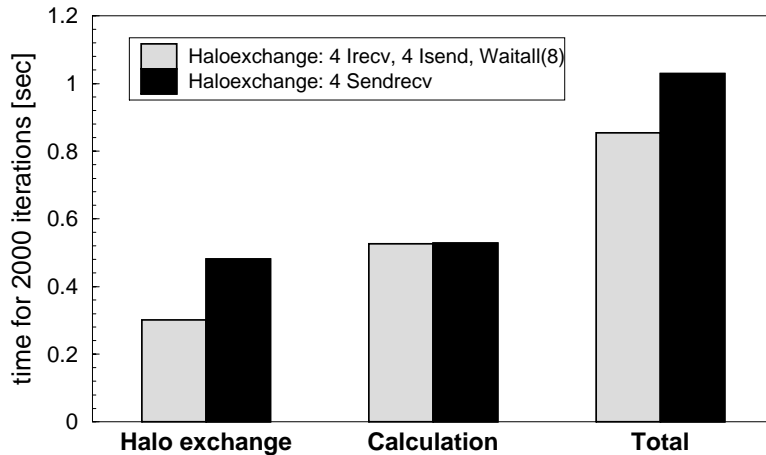


Figure 7: Using non-blocking communication for the Jacobi inverter. Results are for 2000 iteration on 1024 processors from 128 LPARs. The problem size is 6720×8064 .

formance very close to the limit of the switch adapter pair for messages of 16 kB. This is in contrast to the performance of the blocking patterns, finishing a single communication at a time. These patterns are generalisations of Patterns 1 and 2 in the previous section. For small message sizes, their performance is essentially flat, however when sending 16 kB sized messages the performance decreases quite substantially when increasing the number of dimensions.

In summary, with a series of benchmarks we have shown that the performance of the MPI library can be substantially improved by using non-blocking sending and a final call to `MPI_Waitall` for messages of a few kilo-bytes. To benefit from this improvement it is unimportant whether the messages go to the same or different processors.

8 Jacobi inverter in two dimensions

A simple Jacobi inverter of a lattice Laplacian in two dimensions provides us with a “*closer to live*” example for a real application. The performance of this benchmark style kernel has been described in detail in reference [1]. This is a simple example for a field theory using domain decomposition.

In [1] the performance of the code has been investigated using `MPI_Sendrecv` to exchange the halos between the processors. It is interesting to see how much of an improvement one can achieve for this application using the results of this paper. We choose a parameter regime where the code spends approximately equal time in computation and communication. Due to significant super linear scaling in the computational part, the version using `MPI_Sendrecv` in the communication part is still efficient in this regime.

In Figure 7 we report on the performance of this kernel when 4 calls to `MPI_Irecv`, 4 calls to `MPI_Isend` and a call to `MPI_Waitall(8)` are used instead of 4 calls to `MPI_Sendrecv`. Using a problem size of 6720×8064 4-byte floating point numbers on 1024 processors leads to halo sizes of 848 bytes and 1016 bytes. We compare the fastest out of 30 trials for each version. For both versions the calculation performs equally well. We note an improvement of 37% for the halo exchange (communication) when using non-blocking communication which leads to an

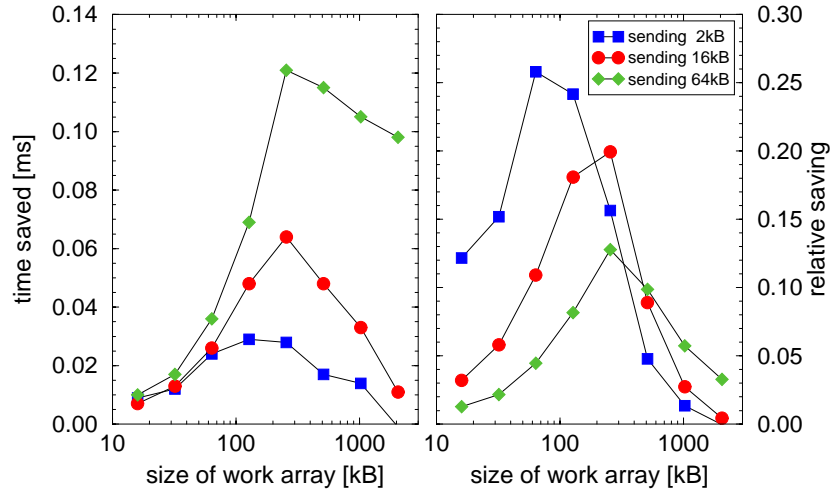


Figure 8: Time saved when overlapping work and communication on a torus in one dimension, when using non-blocking communication.

improvement of 17% for the total execution time of this version of the application.

9 Potential for overlapping calculation and communication

Before concluding, we want to estimate the potential for overlapping calculation and communication when using the non-blocking communication pattern. We created two modified versions of the torus benchmark. Both versions have additional code to add two vectors of 8-byte floating point numbers. In one version this code was included after issuing the all `MPI_Isend` calls and before the call to `MPI_Waitall`. The other version had the calculation added after the `MPI_Waitall`, such that the communication had to finished before the calculation.

In Figure 8 we give the time saved when placing the work before the `MPI_Waitall` instead of after it. The left hand side of the figure gives the absolute saving in ms, the right hand side the saving relative to the total time. To obtain clean results, we did 10 or more trials for each code version and run time parameter set. For each of them, we selected the fastest observed runtime before doing the differences and ratios.

On the left hand side of the figure we show the time saved per iteration. For small work arrays, for obvious reasons, the time saved increases with increasing work size. However it is interesting to note that the time saved decreases again for work arrays larger than a few 100 kB. A possible explanation might be the calculation and communication start to content for entries in the level 2 cache. For the messages size investigated, the potential for time saving increases with increasing message size. On the right hand side we show the relative saving with respect to the total execution time. The graph indicates, for small messages of a few kilo-bytes there is a potential to save well over 20% of the total execution time by overlapping calculation and communication and even for messages as large as 64 kB the savings can be more than 12%. However as the graph also shows, these savings depends strongly on the size of the work.

The last question we address is whether the overlap of work and communication can be in-

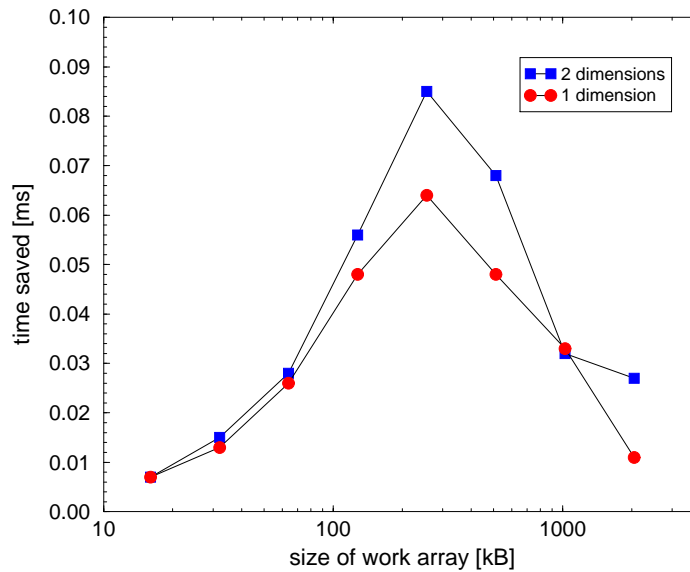


Figure 9: Overlap of communication and computation on a torus in 1 and 2 dimensions. The figure gives the time saved when putting the work before the MPI_Waitall for different sizes of the work array. The torus extends for 4 processors from different LPARs in each dimension. The message size is 16 kB.

creased when a different number of non-blocking communication calls are overlapped. This has been investigated by changing the number of dimensions of the torus when sending messages of 16 kB. The result shown in Figure 9 indicates that there is indeed a greater potential for time savings when more non-blocking calls are overlapped.

10 Summary

In a series of benchmark codes we showed that point-to-point communication on the HPCx system with messages in the kilo-byte range between a number of processors can be implemented most efficiently by using non-blocking communication. The associated wait calls should be issued only after the last message has been sent. Compared to sending single messages, this proves to be superior with respect to latency and makes better utilisation of the dual plane switch architecture. The halo-exchange in a domain decomposition code is a typical real life example whose performance can be improved by using such a communication pattern.

We further showed that there is potential for overlapping communication and calculation, even when using such a dense pattern of communication. The potential for overlapping communication and calculation appears to increase when overlapping the calculation with a larger number of non-blocking communication calls.

Acknowledgement

The HPCx service is funded by the UK Government through the Engineering and Physical Sciences Research Council (EPSRC). We would like to acknowledge useful discussions with Lorna

Smith.

References

- [1] Joachim Hein, Mark Bull, “*Capability Computing: Achieving Scalability on over 1000 Processors*”, Technical Report HPCxTR0301, available: www.hpcx.ac.uk/research/hpc