

Planned AlltoAllv

A Cluster Approach

Adrian Jackson and Stephen Booth

July 12, 2004

Abstract

The MPI AlltoAllv operation is generally based on a large number of point-to-point messages being sent by all the processes in the communication. This report aims to discover whether this process can be optimised for a *clustered* architecture by collating the data from all the processors within a *node* into a shared memory segment and then sending all the data for one node using a single message. This planned method has the potential to decrease overheads by greatly reducing the number of messages sent, and therefore diminishing latency, of the AlltoAllv call. We compare the performance of the planned AlltoAllv with that of the standard MPI AlltoAllv on a number of clustered architectures, and find that for a standard benchmark the planned AlltoAllv can perform considerably better than the MPI version, but for a more realistic benchmark the performance benefit is not as great.

1 Introduction

MPI's AlltoAll and AlltoAllv [1] are collective communication operations that allow all processes involved to exchange some data. AlltoAllv enables all processes in the communicator to send an individual vector of data units (of specified length) to every other process. The lengths of the data vectors sent to each process can be of different sizes, but the datatypes must be the same. The AlltoAll function is a specialisation of the more general AlltoAllv operation, where processors are restricted to sending vectors of the same length.

The AlltoAll operations are often used in scientific codes, especially in codes that involve performing FFTs or global data transposes, parallel quicksort, and array distribution. However, because they are collective operations involving communications from all processors, they can be very time consuming. For example, Ashworth et al [7] demonstrated that collective and global operations limited the performance of a series of applications on a clustered SMP system, and demonstrated significantly better scalability and performance when these operations were optimised or removed.

A standard way to implement AlltoAll operations is for every processor to perform n send and n receives (if there are n processors in the communication), and there is scope for optimisation using non-blocking functions, and ordering communications. This implementation may be sufficient for purely distributed-memory machines, however, the current trend in HPC technology favours shared-memory clusters.

When performing MPI on a shared-memory cluster, there is scope for optimisation of communications within the shared memory node, using fast memory to transfer data rather than (relatively) slow networks. Vendors of shared-memory cluster generally have optimised their MPI libraries to take advantage of this performance benefit for MPI communications.

There have also been numerous research efforts to improve the performance of AlltoAll operations. Thakur and Gropp [6] had mixed success optimising the standard MPICH AlltoAll operation (which is based on nonblocking

sends and receives) for a cluster shared-memory system by ordering the exchanges between pairs of processors to take advantage of the switched network they were using. Their optimisation worked for short messages, but was less successful for larger messages, and they planned to extend their optimisations to exploit the shared-memory nature of the nodes in a cluster system. Hempel et al [8] also attempted to optimise collective communications for clustered systems, reducing network contention by restricting the communication pattern of processors within a node of the cluster, overlapping local and global communications wherever possible, and using shared-memory communications.

However, shared-memory clusters open up the possibility of further optimisations for the AlltoAllv operation. Given that fast MPI communications exist within the shared-memory node, the main bottleneck for MPI communications within a shared-memory cluster is the message latency and bandwidth of the network that connects the nodes. For any given AlltoAllv operation we cannot reduce the amount of data that has to be sent, however, we may be able to reduce the number of messages sent between nodes, and therefore reduce the performance impact of message latency.

As mentioned previously, a standard implementation of an AlltoAllv operation involves n messages being sent from each processor (if there are n processors in this collective communication), giving a maximum of n^2 messages (it could actually be $n \times n - 1$ messages if the implementation doesn't use a message to send data from a processor to itself, and much smaller if some of the vectors have zero entries). However, the nature of a clustered system suggests that we should only need one message between every node rather than one message between every processor. If we had a clustered system with 16 processors per node, and 8 nodes, then the traditional AlltoAllv would need a maximum of 16384 messages, whereas a node aware version would only need 64 messages. Therefore, if we can collate all messages from all processors on one node, and order them by the node they are to be sent to, then we can greatly reduce the number of messages sent.

In order to perform this collation of messages we would need to perform some precomputation and communication of data between processors, to precompute a *plan* of the messages to be sent in the AlltoAllv operation. This plan, similar in nature to the plans used by FFTW [3], is designed to be precomputed once and used many times, as the plan computation will take a significant amount of time and may involve additional communications.

2 Systems

The main platform for the development and testing of this code was HPCx¹. At the time this report HPCx has two systems, one a 1280-processor IBM Power4 (p690) cluster with 8-processor nodes and using the SPswitch2, the other a 640-processor IBM Power4+ (p690+) cluster with 32-processor node and IBM's High Performance Switch (HPS). These are referred to as p690 and p690+ respectively in this report. The p690 cluster has 1.3GHz CPUs, and 8GB of RAM per node. The p690+ cluster has 1.7GHz CPUs, and 32GB of main memory per node.

A number of other systems were used to test this code, including APAC's (Australian Partnership for Advanced Computing National Facility) AlphaServer SC, a 508-processor Alpha ev68 system comprised of 4-processor nodes and connected by a quadrics Elan3 network, and EPCC's Sun system, Lomond, a Sun Fire 15k server with 52 UltraSparc III processors. The Alpha system has 1GHz CPUs and between 4 and 16GB of RAM per node.

The majority of the code development was performed on the p690 cluster. The code was developed in C, using the compilers and tools available on the systems mentioned above.

3 The Algorithm and Implementation

There are two parts to this planned AlltoAllv, the plan computation, and the data communication. Together they are intended to replace the MPI AlltoAllv which has the following specification:

¹This is the UK's National HPC Service, and is supported by a consortium of EPCC at the University of Edinburgh, Daresbury Laboratory, and IBM. The project is funded by the Engineering and Physical Sciences Research Council (EPSRC).

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype,
void* recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

We intend that our AlltoAllv use only the information provided to the standard MPI call, allowing the MPI AlltoAllv function to be easily replaced.

3.1 Plan Computation

The planning stage aims to perform any computation and communication that can be done once and reused for each global communication. In order to fully optimise the communication phase it is desirable for all processors to know what messages will be sent and received in the AlltoAllv operation. Therefore, the first stage of the plan involves each processor calculating how much data it will send to every other processor. This can be derived from the `sdispls` array and the `sendtype` datatype. Once each processor has calculated this, it shares this information with all the other processors in its node. From this each processor can calculate the total amount of data to be sent from its node to any processor in the communicator, and consequently the amount of data to be sent from its node to each of the other nodes in the communicator.

The next step is for all the nodes in the communicator to swap this data. Once this has occurred, each processor can calculate how much data it will be receiving from the other processors (and consequently from the other nodes) in the communication. From this data, all the processors can construct a map of the send and receive spaces for the AlltoAllv communication, allowing them to calculate the layout for the send and receive buffer that will be used to hold the data that will be sent and received from each node.

These buffers, the send and receive buffers, are the key to the performance the communication. They are implemented as *shared-memory segments* [5], meaning only one buffer needs to be allocated for each node, and all processors can access it.

After each processor has worked out where it will place its send data, and where it will be getting its receive data from, as well as where this data should come from and go to (i.e. the displacements in the send and receive buffers, `sendbuf` and `recvbuf`, passed to the function), the next task is to determine which processors will be performing the send and receive operations. For any given node, we need a processor to send each node message, and processors to receive the messages from the other node. To this end, processors are assigned a node to send to, and a node to receive from. This assignment is done in such a way as to ensure that, as long as there are enough processors in a node, each processor either has one message to send or one message to receive. Only if twice the number of nodes is greater than the number of processors within a node will a processor have more than one task to do. This method attempts to load-balance the sending and receiving of messages, both in terms of computation and communication. As well as calculating this list (of sending and receiving processors) locally, it also has to be swapped globally to ensure all nodes know which processors they should be receiving from, and which they should be sending to.

All the planning data is stored in a data structure, which can be passed to the communication function, and reused whenever necessary.

3.2 AlltoAllv Communication

Given the precomputed plan, there are only three operations needed to perform an AlltoAllv communication. The first step is for each processor to packed their data into the shared memory buffer. This is done using the precomputed data in the plan which shows each processor where the data to be sent should be taken from in `sendbuf` and placed in the shared-memory segment.

Once the data has been packed then it can be sent. Each node's data is sent as a single message, by a single processor, using the send/receive pattern already defined in the plan. The messages are sent using MPI non-blocking, asynchronous, communications, with data both being sent straight from the shared memory segment, and received directly into the shared memory segment. As each message sends or receives distinct data, this can be done without encountering any shared memory locking issues.

After the send/receive process has finished, the final step is for each process to extract its particular data from the receive section of the shared memory segment, and populate its receive buffer (`recvbuf`). Because the location of each processor's data within the shared buffer has already been computed, this is a straightforward process.

3.3 Additional Functionality

As well as planning and communication, there are a number of sundry functions that are needed to perform this planned AlltoAllv. Initialisation and finalisation functions are needed to setup and destroy the shared memory segments, and MPI communicators, used. The functions are designed in such a way as to only need one shared memory segment for all the communications, no matter how many plans are created. The same shared memory segment is reused whenever necessary, ensuring that memory is conserved.

Another function that is needed to manage memory is a plan destroyer function. When a plan is created, the plan structure consumes a significant amount of memory. To ensure that this memory is correctly deallocated when the plan is no longer needed, we created a function to free all memory associated with a given plan.

4 Benchmarking

The first method of measuring the performance of the planned AlltoAllv was using a "pure" benchmark. This involved each processor sending the same length vector to each other, first using the planned AlltoAllv, then using the MPI AlltoAllv. Each function was called a number of times, with that grouping of functions itself being repeated a number of time. An average time is then calculated for the group of functions. The benchmark was repeated for different vector lengths, and different processor numbers.

The benchmarks were conducted using vectors of doubles of the following lengths: 1, 5, 10, 20, 40, 80, 160, and 320.

4.1 p690 Performance

For the p690 machine, the benchmarks were conducted using 8, 16, 32, 64, 128, 256, and 512 processors (i.e. 1, 2, 4, 8, 16, 32, and 64 nodes). The results from these benchmarks are shown in Figure 1, which contains all the results collected, and Figure 2, which displays some of the results separately.

We can see from Figure 1 that the Planned AlltoAllv consistently outperforms MPI's AlltoAllv, both as the number of processor, and the size of the data vector, is increased. In fact, the planned AlltoAllv is considerably better, range from twice to one hundred times quicker. Figure 2 shows a selection of these results, and they highlight the fact that the performance difference is greater when the data vector is the smallest (i.e. when using 1 Double) and when the number of processor used is large. However, even when used within a single node, the planned version outperforms IBM's implementation of the AlltoAllv.

4.2 p690+ Performance

The benchmarks for p690+ (which has 32-processor nodes) were conducted using 1, 2, 4, 8, and 16 nodes (i.e. 32, 64, 128, 256, and 512 processors). Again, the results from these benchmarks are shown in Figure 3, which contains all the results, and Figure 4, which displays a selection of results separately. We would expect the planned AlltoAllv to perform well on the p690+ cluster as it has "fat" nodes, meaning that the planned version will considerably reduce the number of messages being sent.

However, IBM's MPI implementation for Phase 2 has been optimised for shared memory communication within a node, so we would not expect the planned AlltoAll to perform considerably better than the MPI AlltoAllv for small numbers of processors.

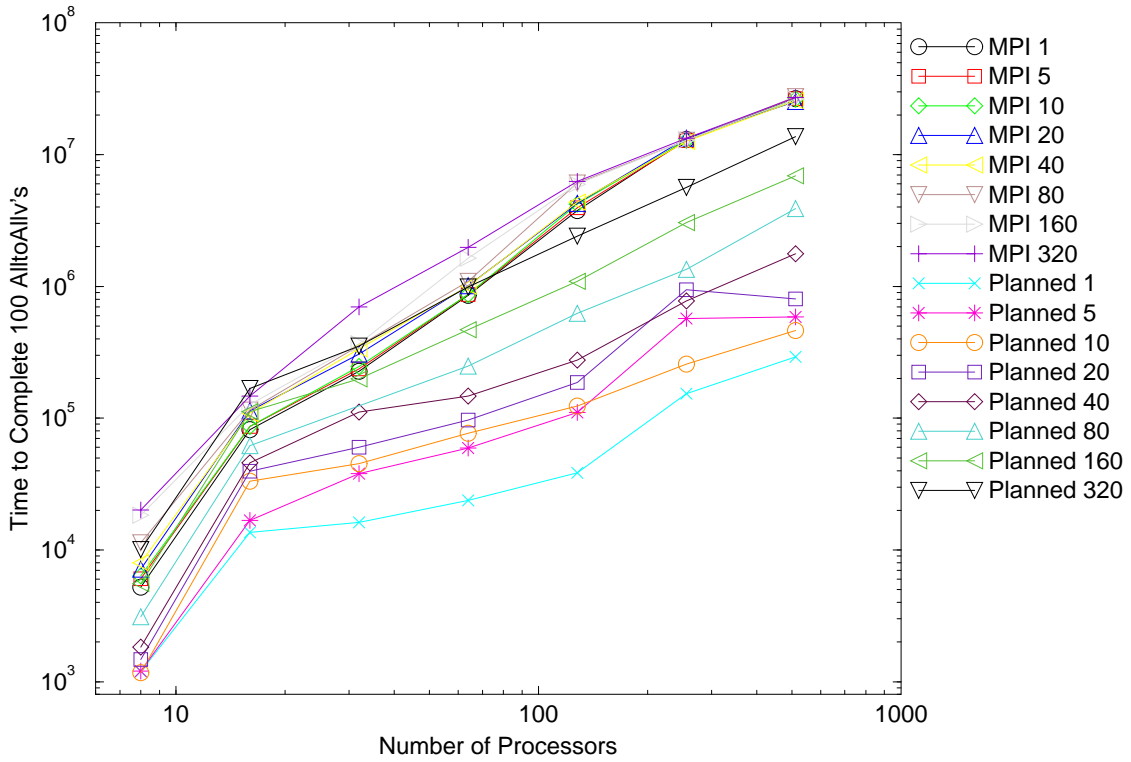


Figure 1: Time for AlltoAllv's using a Uniform Number of Doubles on the p690 cluster

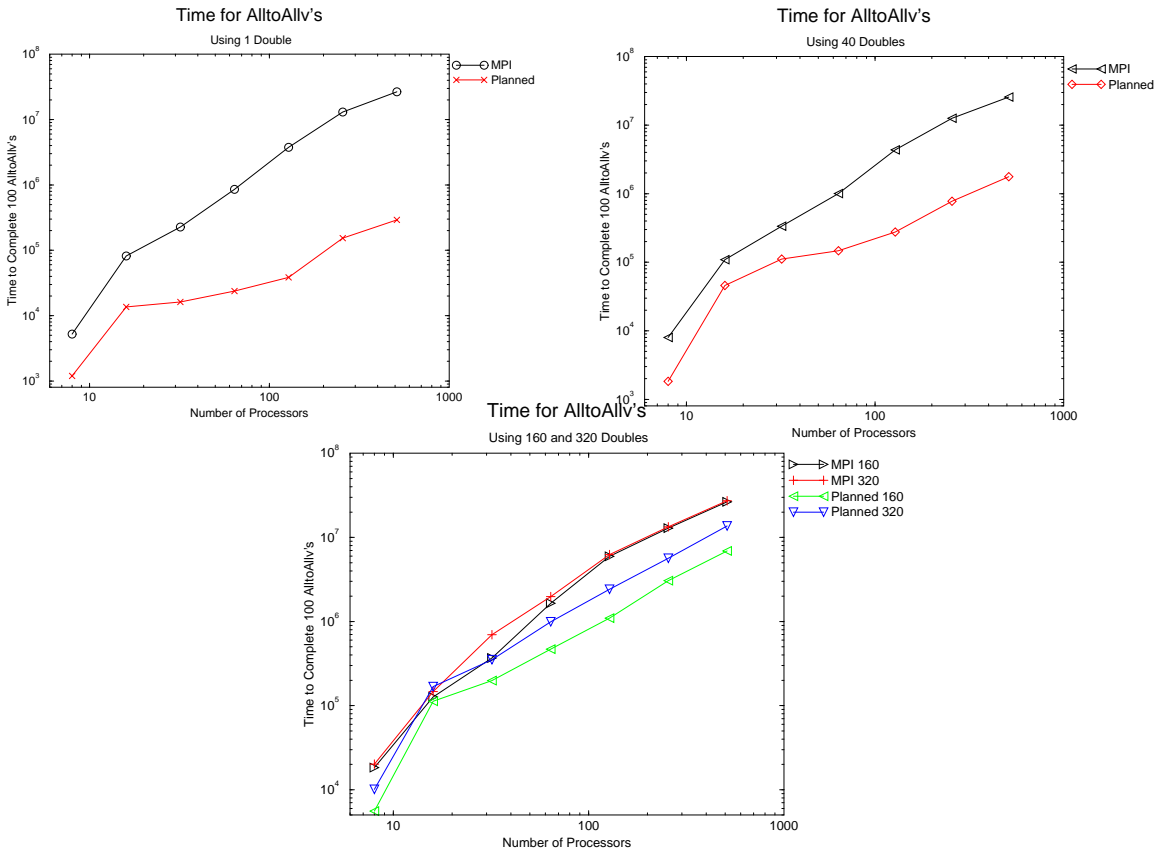


Figure 2: A Selection of Performance Graphs for the p690 cluster

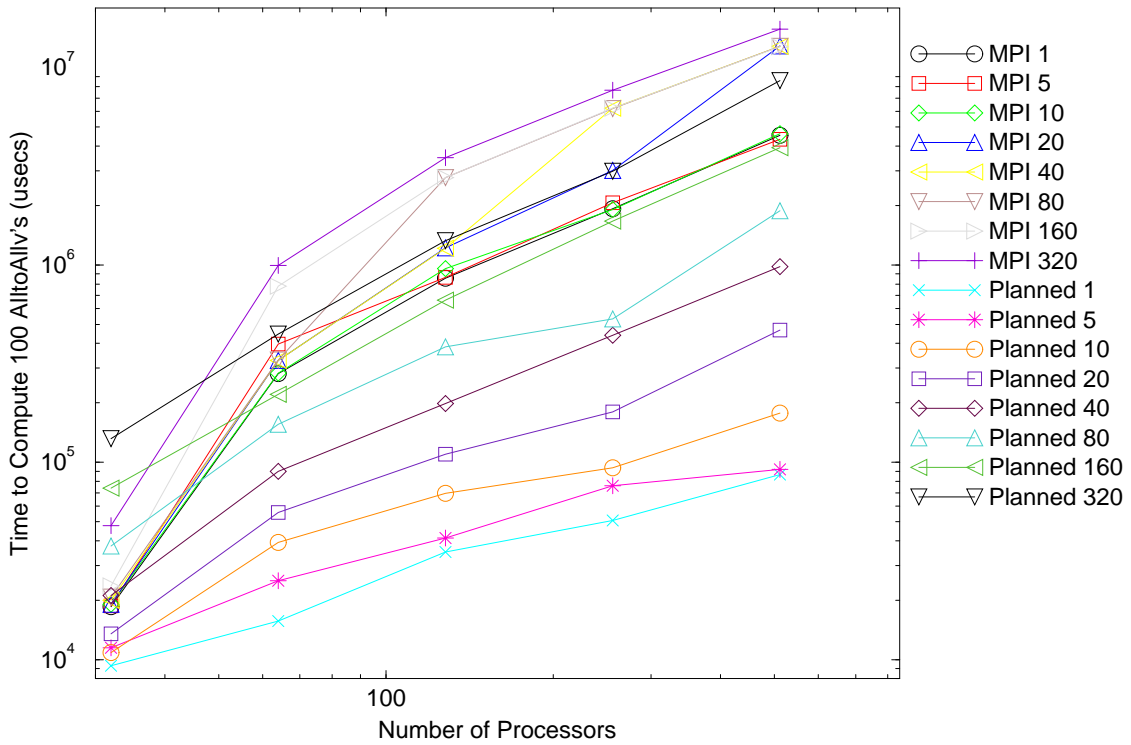


Figure 3: Time for AlltoAllv's using a Uniform Number of Doubles on the p690+ cluster

We can see from Figure 3 that, as with the p690 cluster, the planned AlltoAllv is considerably faster than the MPI version. However, for 1 node times (i.e. 32-processor runs), the performance of the MPI version is often comparable with, if not better than, the planned version. This is due to the optimised MPI library mentioned previously.

It is also evident from Figure 4 that the performance benefit gained by using plans is greater on the p690+ cluster than on the p690 cluster. The performance difference between the the planned and MPI version using 40, 160, and 320 doubles is greater than for the same data sets on the p690 cluster.

4.3 AlphaServer SC Performance

The AlphaServer SC system has smaller nodes than both of the HPCx machines, and a limit of 64-processors per job. Benchmark were carried out on 4, 8, 16, 32, and 64 processors (1, 2, 4, 8, and 16 nodes). Again, Figure 5 shows the full set of results, and Figure 6 a number of relevant individual results.

Because the nodes are "thinner" than the HPCx machines, we would expect the performance gain to be limited. Figure 5 shows that, again, the planned version performs considerably better than the MPI version, and in fact the thinness of the nodes does not seem to impact the performance gain.

As with the HPCx results, we can see (from Figure 6), that the biggest performance benefit is from small vector sizes. It would appear from the results that the MPI implementation used here is not optimised to use shared memory communication within nodes. It may be that we do not see the reduced performance for the planned AlltoAllv because the "network" performance of this machine is (comparatively) worse than the network performance of the HPCx machines, meaning the message reduction gained through using the planned version has a larger impact. However, this is not supported when reading the documentation of both systems. Both present very similar performance statistics for their networks.

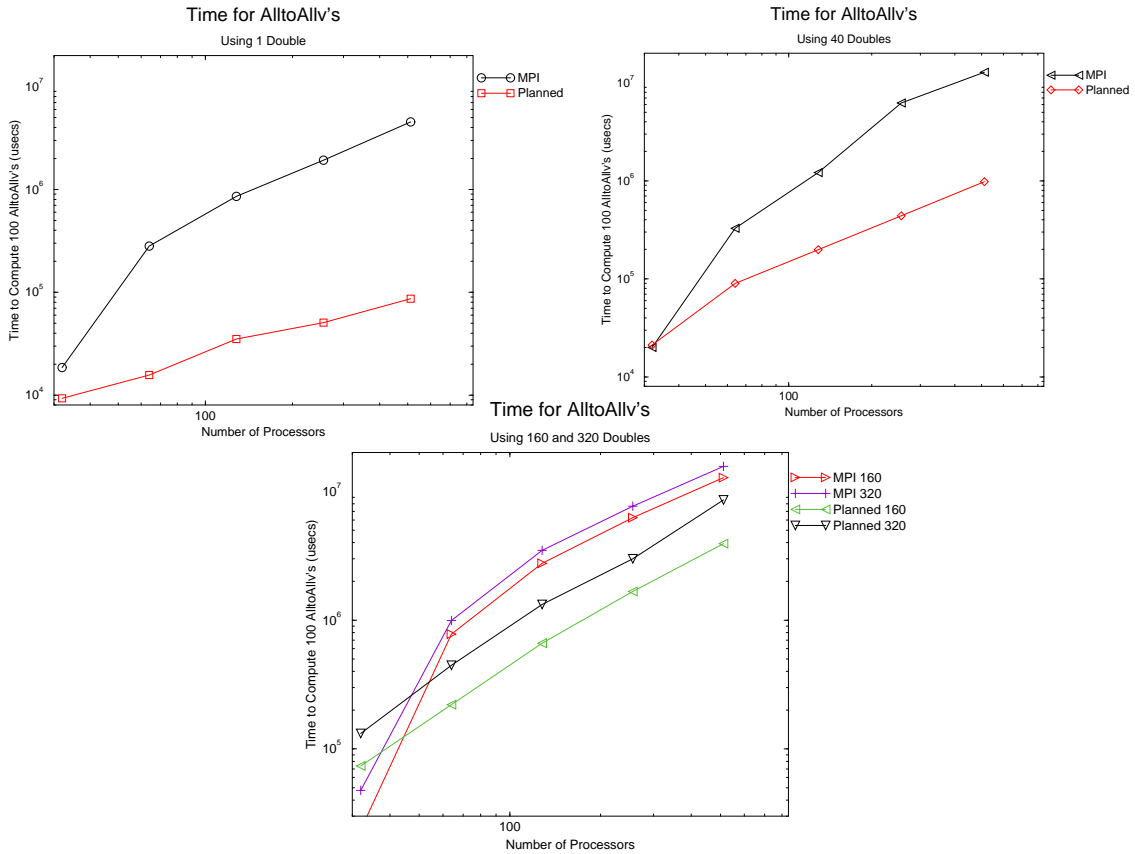


Figure 4: A Selection of Performance Graphs for the p690+ cluster

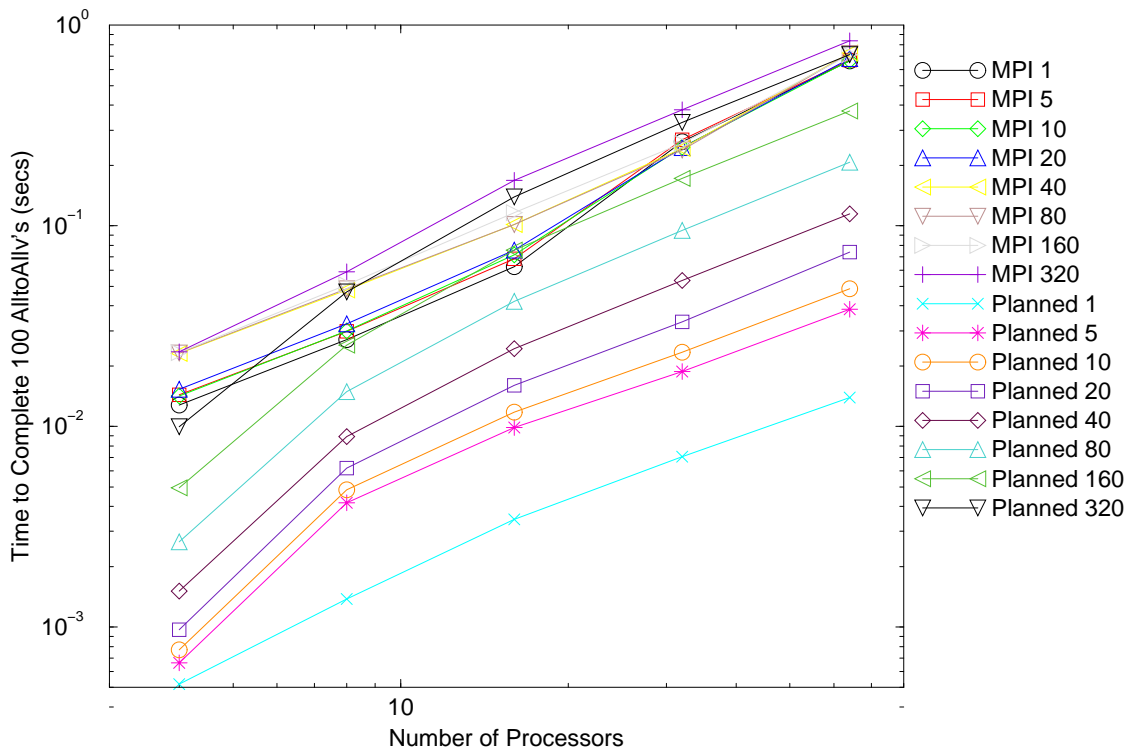


Figure 5: Time for AlltoAllv's using a Uniform Number of Doubles on the AlphaServer SC cluster

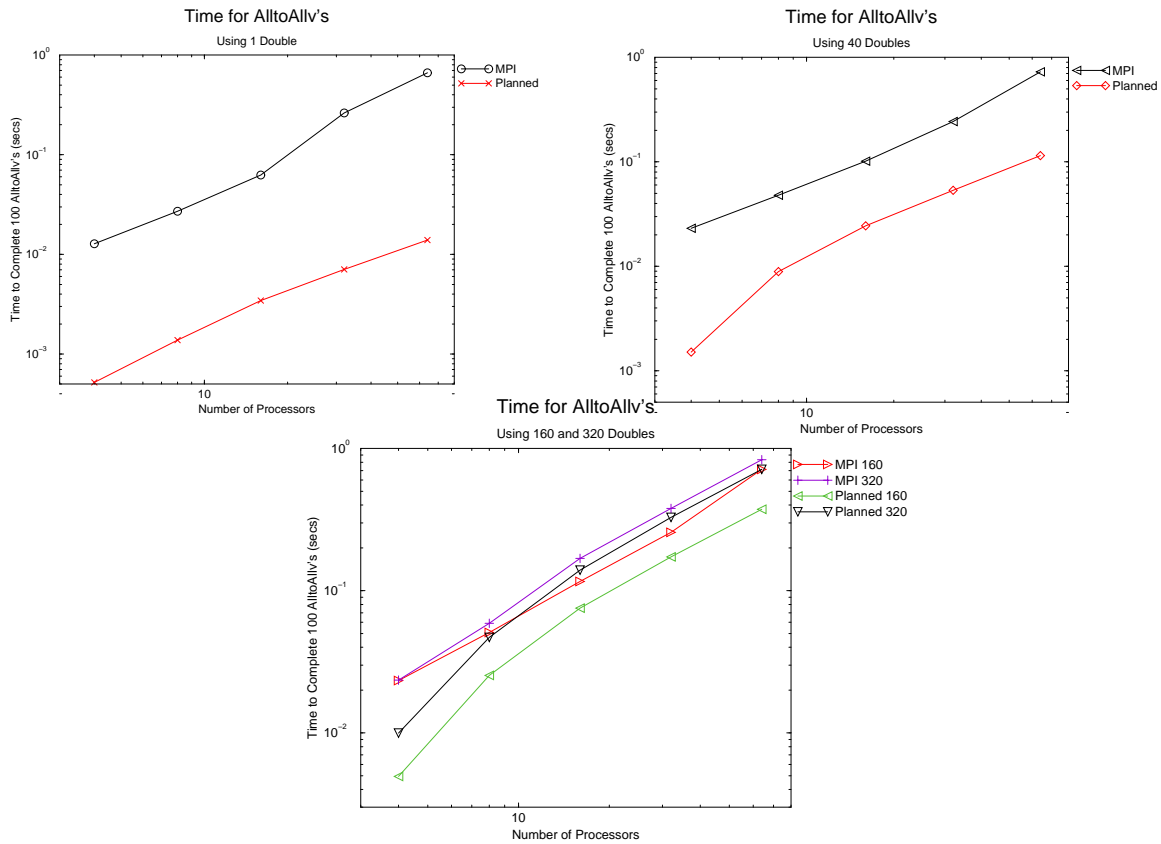


Figure 6: A Selection of Performance Graphs for the AlphaServer SC cluster

4.4 Benchmarking Performance Summary

The results from all three machines show that the planned version of the AlltoAllv can have significant performance benefits when used on clustered systems. The smaller the message used, and the larger the number of processors, the greater the performance benefit. It also shows, however, that for large enough messages we lose the performance benefit. This is because for very large messages, the message latency becomes insignificant compared with the overall cost of sending the message, and therefore the extra work performed to send the planned AlltoAllv data becomes a performance hindrance. Also, when messages become very large on the HPCx systems the MPI sending method becomes strictly synchronous, in which case the MPI AlltoAllv will get a performance benefit from sending lots of smaller message (i.e. messages that do not have to be delivered strictly synchronously) compared to the planned AlltoAllv.

5 Application

Whilst the first method of benchmarking is perfectly valid, and accurately captures performance for an idealised AlltoAllv function call, it does not necessarily reflect how an AlltoAllv is used in real codes. When called in real world applications, AlltoAllv may use vectors of varying lengths from each processor and, more importantly, may contain zero length vectors. As the performance improvement of the planned AlltoAllv is gained from reducing the number of messages sent, if, for example, this is reduced in an actual call by having some zero length messages, the performance benefit seen previously may be reduced.

To test the planned AlltoAllv with more realistic message patterns, an FFTW benchmarking code was used [4]. This code performs benchmarking of the FFTW parallel FFT calculations, which themselves call MPI AlltoAllv. For this

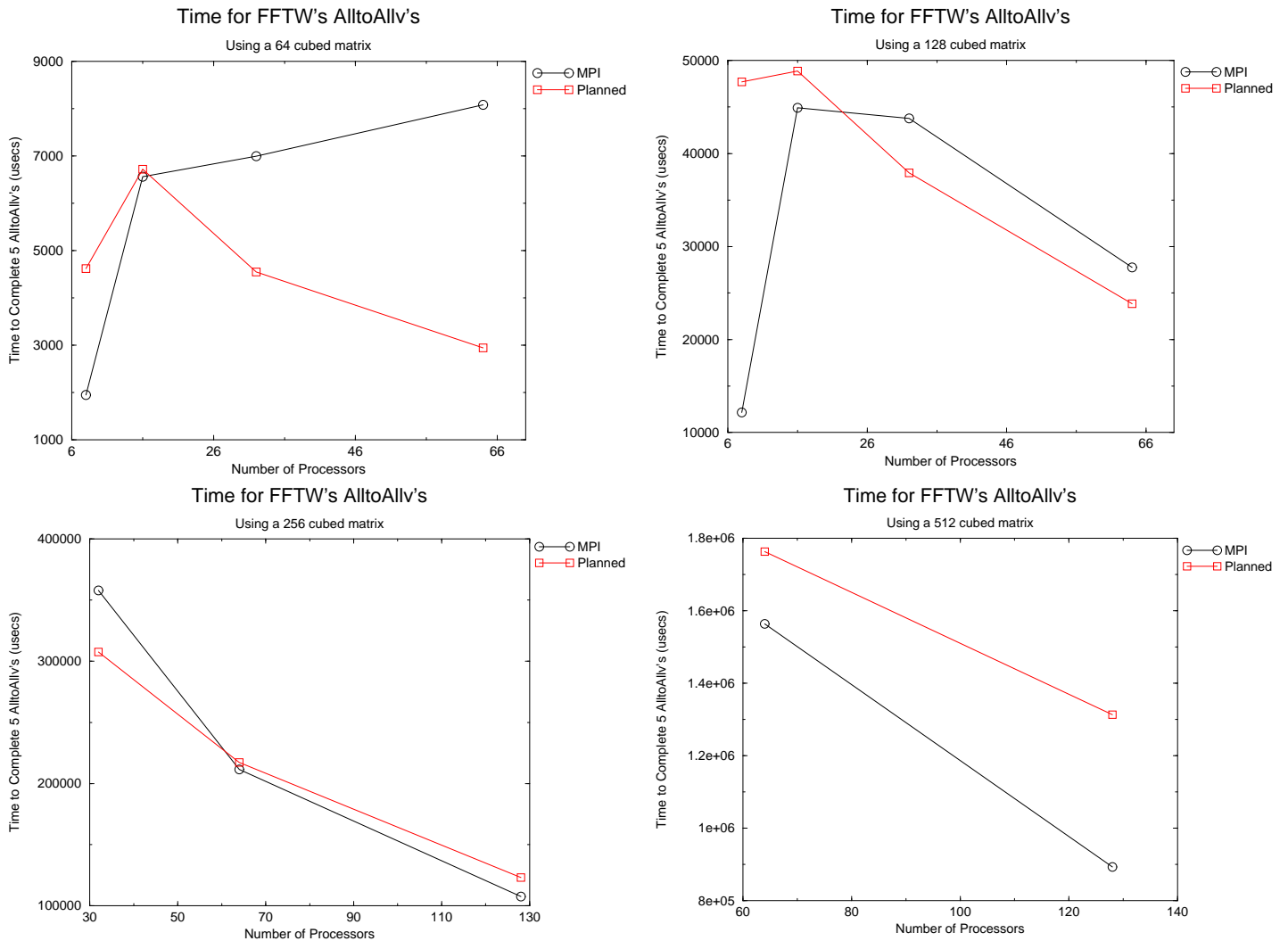


Figure 7: p690 Cluster Performance for Application Benchmark

benchmark, the FFTW library was compiled containing both the MPI AlltoAllv function, and our plan AlltoAllv function, and the benchmark was run using a range of matrix and processor sizes.

The FFTW function used was `fftwnd_mpi`, from the 2.1.5 version of the library. This performed a complex 3-dimensional transform, and was used with matrices of size 64^3 , 128^3 , 256^3 , and 512^3 . These matrices, and the FFTW plans needed for them, consume considerable amounts of memory, which restricted the use of the planned AlltoAllv (which itself requires a reasonable amount of memory, both for the shared memory segments and the plan).

5.1 p690 Performance

The benchmarks were run on a selection of processor sizes, and the results are shown in Figure 7. The first point to note is that, for a given matrix, as the number of processors is increased the amount of data one each processor decreases. This is because the matrix is decomposed (in the first dimension) across the processor available. Therefore, as the planned AlltoAllv has been observed to perform well for small vector sizes and large processor counts, as the processor count is increased we would expect the benefit gained from using the planned AlltoAllv to increase too.

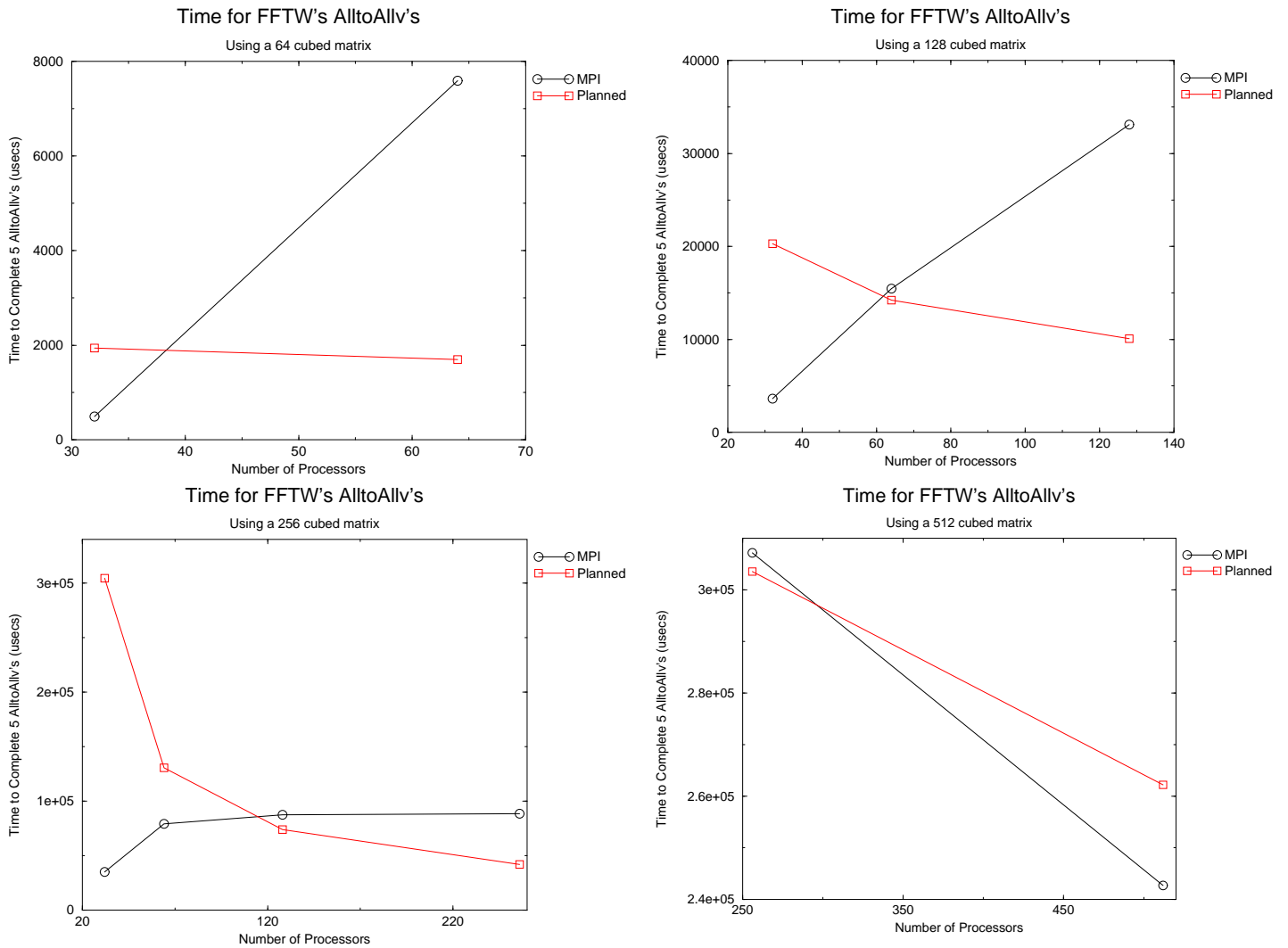


Figure 8: p690+ Cluster Performance for Application Benchmark

The graphs in Figure 7 show that the performance benefit given by using the planned version of AlltoAllv is much less significant in most cases. We can see that for the two small matrix sizes, the planned version is generally better, apart from when only one nodes is used (i.e. only shared memory communication). The larger the matrix, the poorer the performance of the planned function.

5.2 p690+ Performance

The p690+ performance has similar characteristics to the p690 performance, except the point at which the planned performance degrades is at a higher number of processors. This is to be expected as the nodes in the p690+ system are 4 times the size of the p690 nodes

The performance is shown in Figure 8, where we can see that, using 128³ matrix, the planned AlltoAllv is significantly quicker than the MPI AlltoAllv for 128 processor, but has very similar performance for 64 processors. We can also see that IBM's optimised MPI library (optimised to use shared memory communication) has a real performance benefit for the MPI AlltoAllv, which consistently outperforms the planned AlltoAllv when using only one node (i.e. using 32 processor).

5.3 Application Performance Summary

The performance of the standard MPI implementation is, comparatively, much better for this *real-world* application. This can be explained by a number of factors. Firstly, when transposing data in a distributed FFT calculation such as the one performed by FFTW, large sections of the data vector often are empty. That is to say, a lot of the processors involved in the communication are not sending data to some or all the other processors in the communicator. Most processor only usually send data to a small fraction of other processors. A well implement MPI AlltoAllv function will recognise these sends as null operations and not perform them. This means that, overall, there are much fewer messages being sent by the MPI AlltoAllv operation. As the planned AlltoAllv gains its performance benefit by reducing the number of messages sent, this potential benefit is cut when null sends are involved.

Secondly, although a significant number of the data vectors contain zeros, the element that are sent are considerably larger than MPI_DOUBLE's. The matrices FFTW use are a custom data type, `fftw_complex`, that contain storage for at least two doubles, and possibly considerably more internal data. Therefore, the AlltoAllv's called by FFTW have two disadvantages for the planned AlltoAllv; they contain zero length vector elements, that cut the performance benefit of reducing the number of messages, and they contain large messages, which we already know reduce the performance of the planned function.

6 Summary

This research has shown that it is possible to produce an optimised AlltoAllv operation, using only information known to a normal MPI AlltoAllv function, by precomputing a plan. The traditional benchmark shows that on clustered systems, reducing the number of messages sent in the AlltoAllv operation can significantly improve its performance. This is true not only of the AlltoAllv operation, but any message passing operation using a clustered system (or at least the clustered systems we used).

However, the application benchmark shows that the performance of collective communication functions such as AlltoAllv is heavily dependent upon the specific parameters used to call the function. It shows that before we can address the problem of improving the performance of collective communications, we need a good understanding of how they are likely to be used, both the communication pattern, and the data to be sent.

The planned AlltoAllv operation is designed for situations where it can be reused many times (i.e. codes with long run times, which repeat the same calculations many times) as the time needed to construct a plan is significant (when compared to the time consumed by an AlltoAllv call, usually taking 4 times as long).

Although this method of optimising AlltoAllv could never be included in the standard MPI library, because there is no way to identify an AlltoAllv call as persistent and it is only by reusing a plan that the method can improve performance, it does allow for an optimised collectives library to be developed. Such a library, which fully utilises the opportunities of the shared memory nature of the nodes in a cluster, could be used to help improve the performance of user codes without users having to perform significant development work.

References

- [1] **MPI: A Message-Passing Interface Standard**, Message Passing Interface Forum, June, 1995
- [2] **MPI-2: Extensions to the Message-Passing Interface**, Message Passing Interface Forum, July 18th, 1997
- [3] **FFTW Manual for Version 2.1.5**, Frigo, M., Johnson, S.G.,
http://www.fftw.org/fftw2_doc, November 2003
- [4] **3D FFTs on HPCx (IBM vs FFTW)**, Jackson, A., Pringle, G. J.,
http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0303.pdf, June 2003

- [5] **Optimised Collective Communications Via Shared Memory Segment**, Booth, S.,
http://www.epcc.ed.ac.uk/~spb/shm_mpi/
- [6] **Improving the Performance of MPI Collective Communications on Switched Networks**, Thakur, R., Gropp, W., Argonne National Laboratory
- [7] **Towards Capability Computing**, Ashworth, M., Bush, I., Guest, M., Sunderland, A., Booth, S., Hein, J., Smith, L., Stratford, K., and Curioni, A., accepted for publication in *Concurrency and Computation: Practice and Experience* (Special Issue), 2004.
- [8] **Algorithms for Collective Communication Operations on SMP Clusters**, Hempel, R., Golebiewski, M., Traff, J., C & C Research Laboratories, NEC Europe Ltd.