View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Publikationen der Universität Stuttgart

Performance of MPI on the CRAY T3E-512

- Test Program
- Point-to-Point Communication
- Barrier Synchronization
- Global Communication
- Global Reduction
- Summary
- Literature

Virtueller Computer durch Vernetzung

Performance of MPI on the CRAY T3E-512

Michael Resch / Holger Berger / Rolf Rabenseifner / Thomas Bönisch

The CRAY T3E-512 is currently the most powerful machine available at RUS/hww. Although it provides support for shared memory the natural programming model for the machine is message passing. Since RUS has decided to support primarily the MPI standard we have found it useful to test the performance of MPI on the machine for several standard message passing constructs.

Test Program

The tests were done using a standard benchmark code that was developed at RUS for the testing of MPI performance on parallel architectures. The code is entirely written in FORTRAN77 to ensure portability. It is intended to test performance of simple point-to-point communication, barrier synchronization, global communication and global reduction operations. Measurements have shown that it makes no significant difference whether MPI is used in C or FORTRAN codes on the CRAY T3E.

The method applied for measurement is to synchronize processes involved in the measurement, call the function measured in a loop several times to get an average timing and measure the time the loop has taken. As a function for time measurement MPI_Wtime is chosen. This ensures portability of the code across all platforms as long as MPI_Wtime is implemented correctly and without significant overhead. For the CRAY T3E this overhead was measured to be less than one microsecond. All measurements were done for numbers of processors up to 256.

Point-to-Point Communication

MPI on the CRAY T3E can make use of different modes for a standard send and recv. Messages can either be buffered by the system or sent directly. The user may decide about the size of the system buffer by setting an environment variable MPI_BUFFER_MAX. The default value is unlimited. This would imply that messages of any size would be buffered by the system (limited only by the total amount of memory available). If MPI_BUFFER_MAX is set to zero messages are not buffered. This implies that a standard MPI_Send would have to wait until a matching MPI_Recv is posted before sending the data. Not every application can guarantee that this waiting time is short or even zero (if the MPI_Recv is posted before the MPI_Send). However, omitting the system buffering avoids additional

copying of the data and thus reduces the communication overhead. Measurements for an optimal size for MPI_BUFFER_MAX have shown that below 4096 buffering does not have an effect on the bandwidth and times measured. An optimum strategy would therefore be to chose MPI_BUFFER_MAX to be 4096. This avoids idle times for small messages but allows to make use of higher bandwidth for larger messages.

In general the following two estimates hold:

If buffer size is smaller than 4K the time in microseconds for a send is:

$$t = 17 + x/50$$

where x indicates the amount of data to be sent in bytes.

For messages larger than 4K the formula becomes:

$$t = 56 + x/308$$

There was some discussion about how streams may influence the performance of MPI calls. Experiments have shown that at least with our tests we can not see any significant difference in bandwidth with respect to usage of streams. Latency is always in the range of about 15 microseconds. Results for bandwidth are shown in figure 1. Bandwidth up to 300 MB/s can be seen for very large messages.

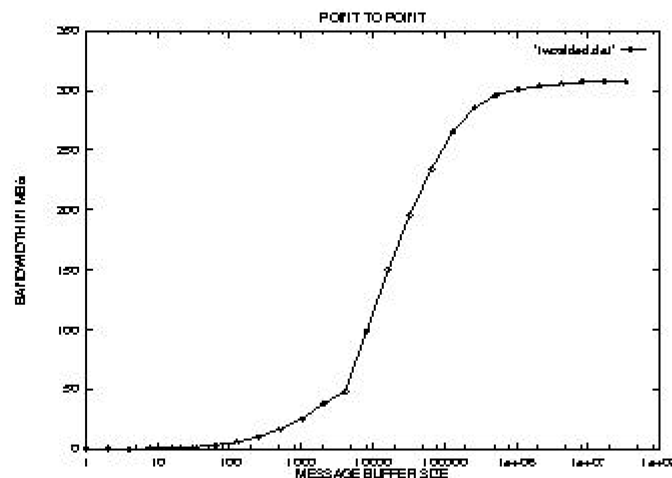

Figure 1: Bandwidth in MB/s for standard send/recv operation for varying message buffer size

Barrier Synchronization

These tests were done to find out about the costs for synchronizing an application. Many applications have points were they have to synchronize according to the algorithm chosen. If these can not be ommitted it becomes important for the code how fast the underlying system can handle the task. The CRAY T3E does no longer provide explicit hardware support for barrier synchronization as did the T3D. However, it provides a clever algorithm to do the synchronization in a distributed way. This is done by forming clusters, synchronizing processes inside such a cluster and then synchronizing the clusters themselves. Synchronization time is given here for a varying number of processes. As shown in Figure 2 synchronization time remains nearly constant up to 64 processes and then goes up to about 8 microseconds for 256 processes. The important thing is that the time does not increase with the number of processes involved which can be observed for some other architectures.

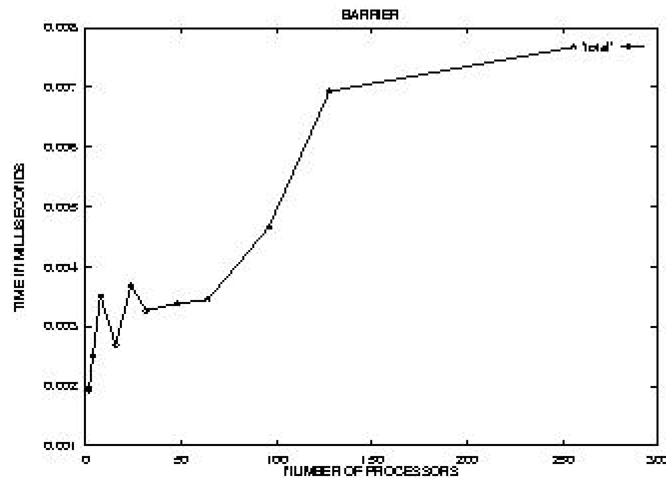

Figure 2: Total synchronization time for varying number of processes

Global Communication

These tests were done to find out about the networks capacity of handling a large amount of communication. Furthermore, global communication is often used in numerical codes. Normally it includes a large number of processes. This means that a lot of data have to be transferred across the machine. The better the network scales with the number of messages sent, the better the performance of global communication.

The basic global communication operation is a broadcast.The tests for MPI_Bcast show that the network of the CRAY T3E scales very well and goes up to a summarized bandwidth of more than 7GB/s on 256 processors and large buffer sizes. In the following two figures buffer size always means the size of the buffer that is distributed from the root process. To get the size of all data that are sent during one call to MPI_Bcast one has to multiply this size by the number of processes involved.

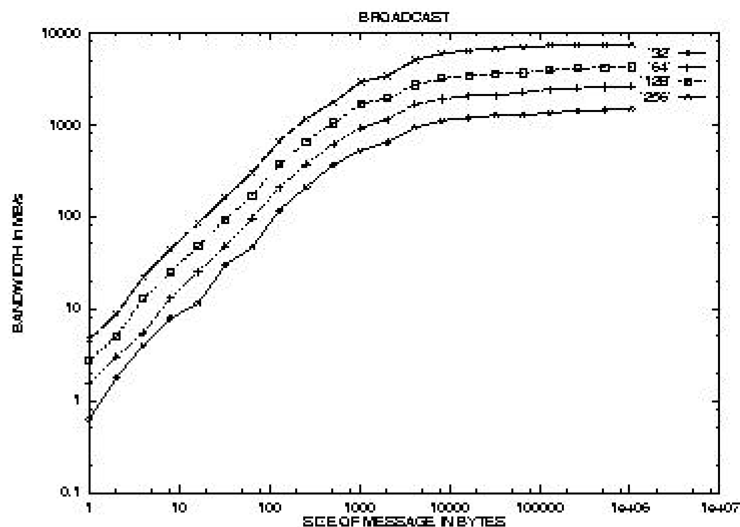

Figure 3: Aggregated bandwidth for an MPI_Bcast for four different numbers of processes for varying buffer size

Both for varying buffer size and for varying processor number the network scales well as shown in figure 3 and 4. As the number of processors involved goes up also the peak bandwidth goes up. And even for small messages in the range of 512 Byte one can achieve good results for typical processor numbers.

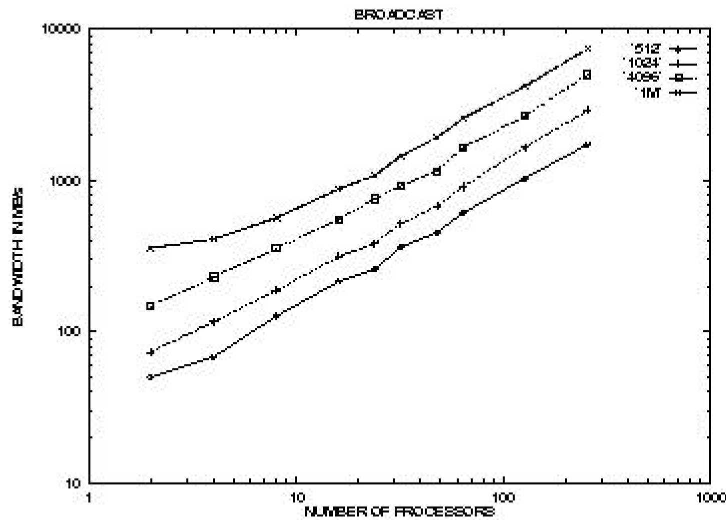

Figure 4: Aggregated bandwidth for an MPI_Bcast for four different message buffer sizes for varying number of processes

An interpretation of these results with respect to a tree algorithm for the fast distribution of data shows that the broadcast performs excellent. For 32 processors a broadcast would need 5 communication steps. For a message of size 4K the resulting time for communication using send and recv would be 0.1401 milliseconds. The result of MPI_Bcast is 0.1385 milliseconds. On 256 processors the result using send and recv would be 0.224 milliseconds. The broadcast result is 0.208 which is again faster. For larger buffers the overhead increases. Distributing 1MB using send and recv would need 16.5 milliseconds but the broadcast actually needs 22.5 milliseconds. Since also the tree algorithm would face overloading of the network this is an acceptable result.In general broadcast is faster than any algorithm the user could implement.

Another interesting feature from the programmers point of view is the all-to-all function that can be used e.g. to transpose a matrix. Again times were measured for both varying number of processes and varying message buffer size. The buffer size given in the next two figures is the amount of data that is sent from one process to the others.

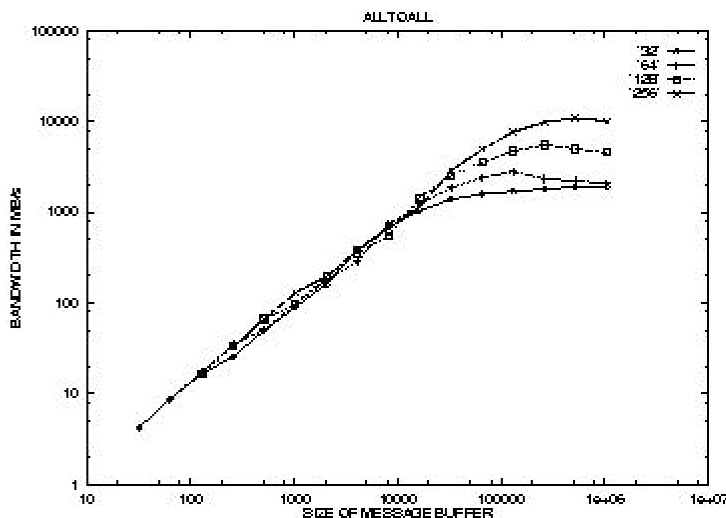

Figure 5: Aggregated bandwidth for an MPI_Alltoall for four different numbers of processes for varying buffer size

To get the total amount of data one has to multiply the given buffer size by the number of processes.

The results in figure 5 and 6 show that reasonable bandwith can already be achieved for small messages

(in the range of 1K) and even for small numbers of processes. To exchange 4K of data the alltoall needs only 0.327 milliseconds on 32 processors.

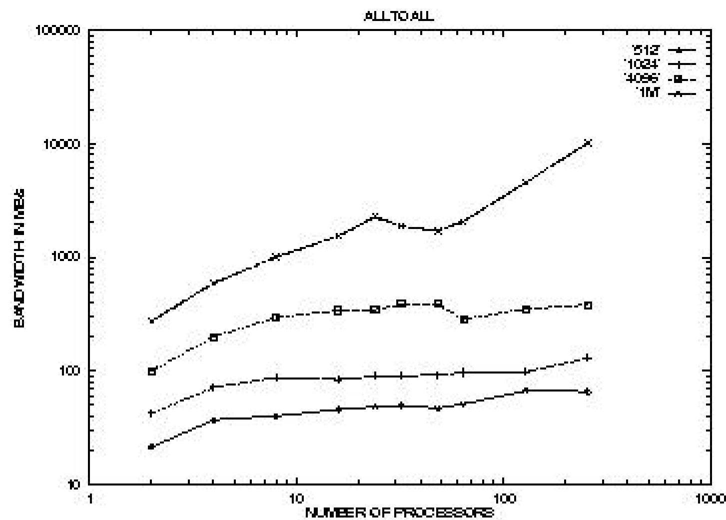

Figure 6: Aggregated bandwidth for an MPI_Alltoall for four different message buffer sizes for varying number of processes

A perfectly implemented algorithm using send and recv would need at least 0.465 milliseconds for the same operation. For medium sized messages the aggregated bandwidth goes already up to 1 GB/s and for very large messages (in the range of 100K to 1M) aggregated bandwidths of up to 2-10 GB/s can be achieved. One has to take into consideration that for a given message size of 1M on 256 processors this means that in total 256 MB have to be transferred.

Another feature that is often used in numerical codes is the gathering and scattering of data. MPI supports this and provides the routines MPI_Gather and MPI_Scatter. For both calls time was measured for varying message size and varying number of processes. In the following buffer size is the total size of the array that is gathered at the root.

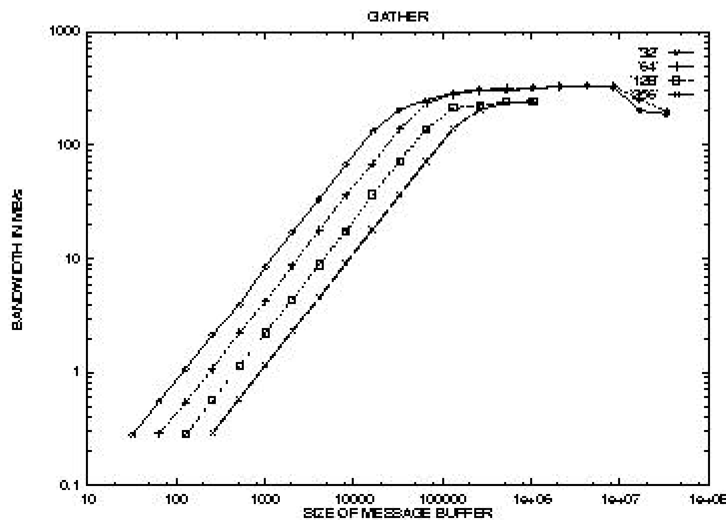

Figure 7: Aggregated bandwidth for an MPI_Gather for four different numbers of processes for varying buffer size

Therefore, it's equal to the total amount of data to be sent. Figure 7 shows that for all numbers of processes investigated aggregated bandwidth remains small for small and medium sized messages and only reaches about 250 MB/s for a message size of 1 MB.

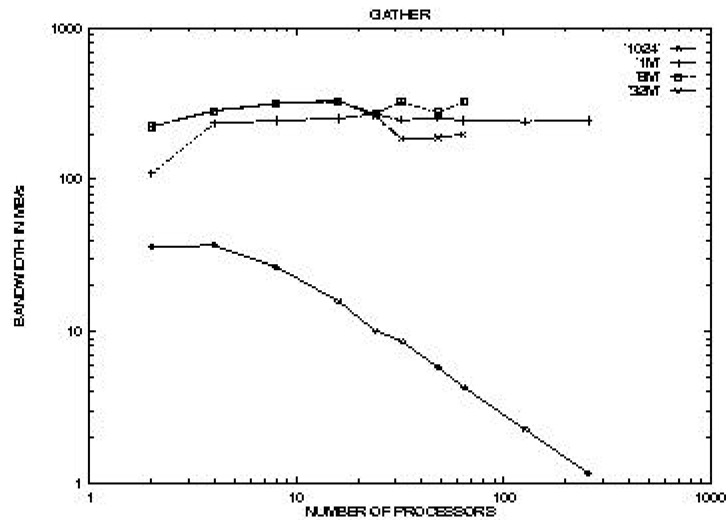

Figure 8: Aggregated bandwidth for an MPI_Gather for four different message buffer sizes for varying number of processes

For larger buffer sizes the bandwidth breaks down again.This indicates that the overhead for handling a gather operation is too high to allow reasonable bandwidth (see also figure 8). The same holds for very small buffer sizes when the system has only to organize the collection of the data while transmitting only some bytes per process. It becomes obvious that the amount of time spent in handling the gather operation increases with number of processes to an extent that gives the impression that a carefully programmed loop for sending the data may even be faster for smaller messages.

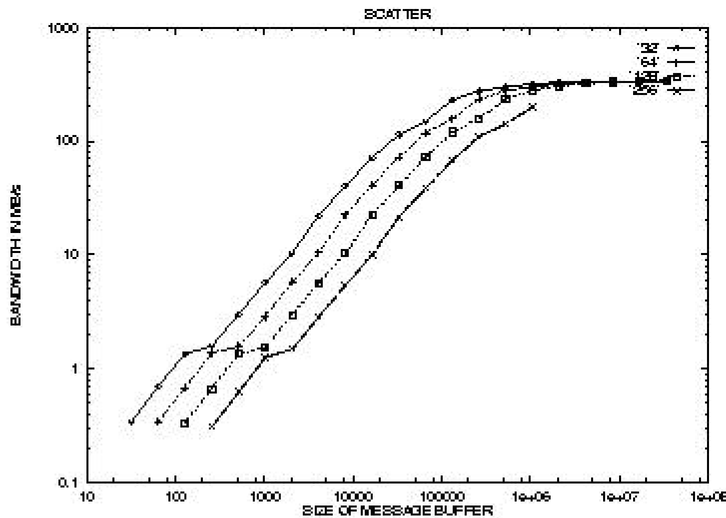

Figure 9: Aggregated bandwidth fo an MPI_Scatter for four different numbers of processes for varying buffer size

The same can be seen for the scatter operation in figure 9 and 10. Again a reasonable bandwidth can only be achieved for large buffer sizes and again for large number of processes the overhead of organising the scatter operation yields inacceptable bandwidths for medium sized buffers.

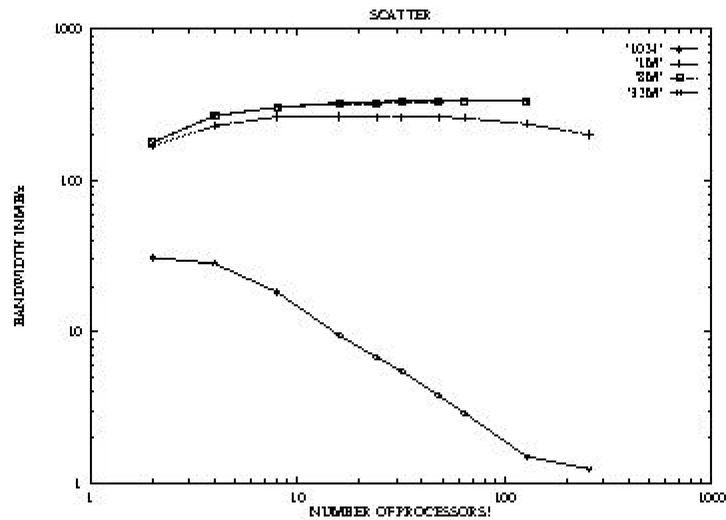

Figure 10: Aggregated bandwidth for an MPI_Scatter for four different message buffer sizes for varying number of processes

Global Reduction

Global reduction is a function that is normally used to collect data to get some global information about a numerical application like a residuum. Since a reduce operation needs both communication and computation only the time for the execution of an MPI_Reduce using the summation (MPI_SUM) is given here.

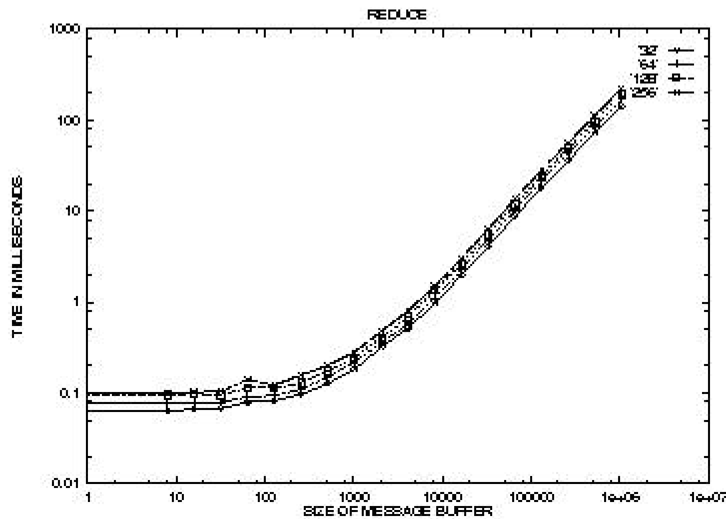

Figure 11: Time in milliseconds for an MPI_reduce for four different numbers of processes for varying buffer size

The results in figure 11 show that for all numbers of processes time goes up for increasing message buffer size. Buffer size now is the length of the array that is used in the reduction function. The total amount of data sent is therefore buffer size times number of processes. The total amount of operations is the same. Since a reduce has to collect data from all processors much like an inverse broadcast we would expect to see similar results as with the broadcast.

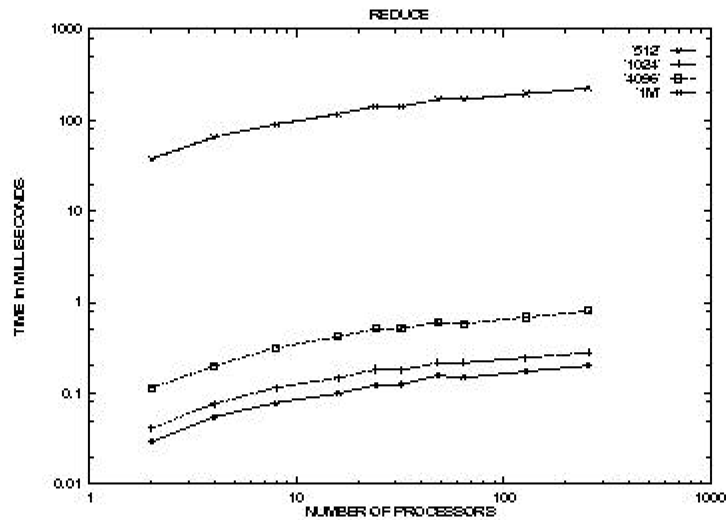

Figure 12: Time in milliseconds for an MPI_Reduce for four different message buffer sizes for varying number of processes

What we can actually see is that time that should be theoretically spent in communication of data plus time for computation of the results is higher than time measured for the reduce operation. This shows that also the reduce operation is implemented well and that a user should better use MPI_Reduce instead of writting his or her own loop to collect such data.

Figure 12 shows that costs scale linearly with increasing number of processors. This is what one could have expected from the results of the broadcast.

The results described above are summarized in the following tables to give an estimate of the costs of all MPI calls described in terms of microseconds. In addition to timings for transfer of data we try to give an estimate of the latency for each operation. We measure this by using a buffer that is of size 8 bytes.The reader should be aware of the fact that one microsecond equals 600 theoretically possible floating point operations. Even if considering that only part of the peak performance of the processor can be achieved one microsecond represents a considerable amount of floating point operations that could be done instead of communication. Costs for a buffer size of 1KB are given in the table below in microseconds.

MPI_CALL	32PEs	64PEs	128PEs	256PEs
MPI_Bcast	35	42	78	90
MPI_Gather	52	108	455	880
MPI_Scatter	88	172	675	816
MPI_Alltoall	171	327	1323	2016
MPI_Reduce	181	217	246	278
MPI_Allreduce	199	253	277	347

The same results for a buffer size of 1MB are:

MPI CALL	32PEs	64PEs	128PEs	256PEs
MPI_Bcast	18920	21575	31793	36133
MPI_Gather	6236	6395	4317	4267
MPI_Scatter	6096	6191	4424	5222
MPI_Alltoall	18085	21885	29315	26491
MPI_Reduce	142858	170491	196300	224312
MPI_Allreduce	163649	191593	223817	256235

Latency for the operations described is:

MPI CALL	32PEs	64PEs	128PEs	256PEs
MPI_Bcast	31.6	38.6	41.52	46.7
MPI_Gather	110.75	229.4	450	877
MPI_Scatter	90.2	184	393	816
MPI_Alltoall	230	472	989	1997
MPI_Reduce	45.8	57.2	94.3	98.2
MPI_Allreduce	85	58.8	114	104.5

Summary

Several measurements have been performed for some basic MPI calls that are frequently used in numerical codes. The results show satisfactory bandwidth and latency for point-to-point communication. Latency is about 15 microseconds while bandwidth goes up to 300 MB/s. Synchronization is also done well and takes only 8 microseconds for 256 processors. The scalability of the network is shown by global communications where an MPI_Bcast can achieve up to 7 GB/s of aggregated bandwidth. For an all-to-all operation results are even better. Already for small messages acceptable bandwidth of about 1GB/s can be achieved and for large message buffer sizes this goes up to 10 GB/s. It has however become obvious that operations like gather or scatter are not able to exploit the networks scalability. On the other hand the reduce operation that is very similar to an inverse broadcast combined with some calculation performs again well.

Literature

[1] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, Version 1.1, University of Tennessee, Knoxville, 1995

[2] Pacheco, Peter, Parallel Programming with MPI, Morgan Kaufmann Publishers, San Francisco, 1997

[3] Berger, Holger, MPI Performance on different Hardware Platforms, Project in course of instruction for Mathematisch-technischen Assistenten at RUS, RUS, 1997.

Michael Resch, NA-5834
E-Mail: resch@rus.uni-stuttgart.de

Holger Berger, NA-2503
E-Mail: berger@rus.uni-stuttgart.de

Rolf Rabenseifner, NA-5530