MPI Performance of a PC cluster at the ICTP

Roberto INNOCENTE
Abdus Salam ICTP/ Sissa
inno@sissa.it

March 22, 1999

The aim of this presentation is to give some figures about performance of MPI in a real installation and on almost up-to-date hardware.
MPI is now an accepted industry standard for message passing. There is a growing number of programs in physics that have been coded with this library on CRAYs T3D/E, IBMs SP1/2, SGIs Origin, etc and that now can be conveniently used on PC clusters *(Refer to Vittoli's presentation)*. We will see anyway in the next paragraphs that there are many subtle points in configuring and installing such a system and that many binary distributions available are tuned for very different environments.

1 Hardware configuration

We installed a cluster of 20 PCs equipped with a Pentium II at 450 Mhz, 384 MBytes of RAM, a Fast Ethernet 3Com 3c905b card and a 4 GB h/d, all interconnected through a 3Com Super Stack II 3300 Fast Ethernet switch *(See Fig. 1)*.

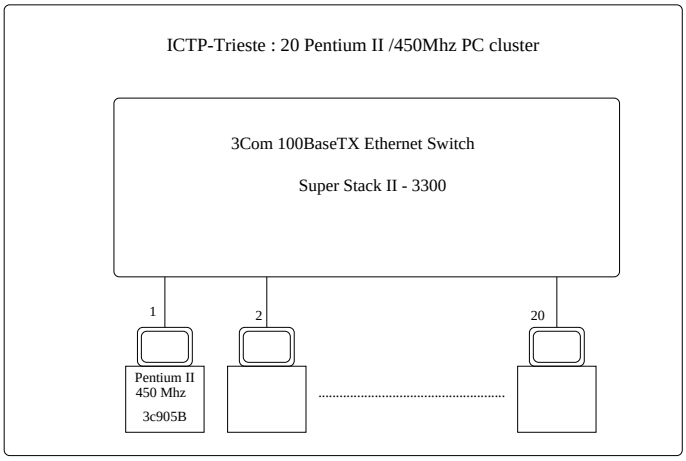

Figure 1: Hardware configuration

2 Software configuration

Among the different MPI implementations we chose Argonne's MPICH because it is widely used, it has a clear interface between a device independent layer and a device dependent one (ADI = Abstract Device Interface) and there are good performance reports for it (over Myrinet a bandwidth of over 110 MB/s and a latency of 7 usec are reported).

3 Memory Bandwidth

We measured the main memory bandwidth with the *stream* benchmark. As you can see these off-the-shelf PCs have a very respectable memory bandwidth: about 300 MB/s *(See Fig. 2)*. This is sufficient to guarantee that we will not

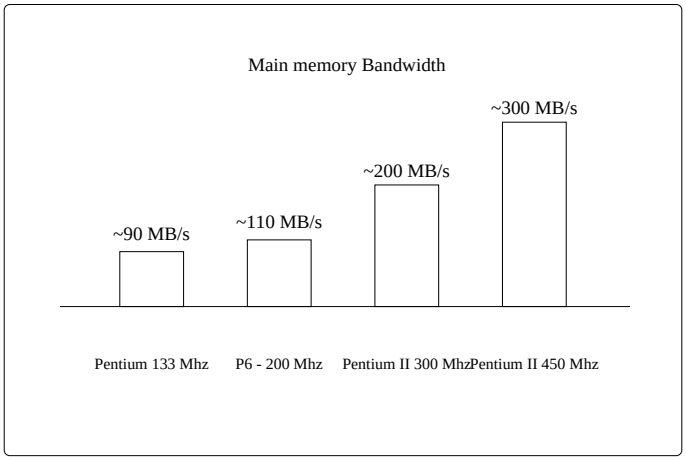

Figure 2: Main memory bandwidth

have problems with the transfer rate involved with one or multiple Fast Ethernet devices.

4 Network performance

We measured the UDP/TCP bandwidth between 2 nodes with standard tools like *ttcp* and *netperf*. It comes out that the UDP bandwidth is about 11.7 MBytes/s and the TCP bandwidth is about 10.6 MBytes/s.
The UDP bandwidth is about 10 percent higher. This is due to the fact that the TCP header is longer and to the higher processing overhead required by TCP *(See Fig. 3)*.

4.1 Using UDP

Using UDP is appealing because of the less overhead and greater efficiency involved. Anyway there are many features of TCP we need to re-implement over UDP if we want to use it as an MPICH abstract device. We need error

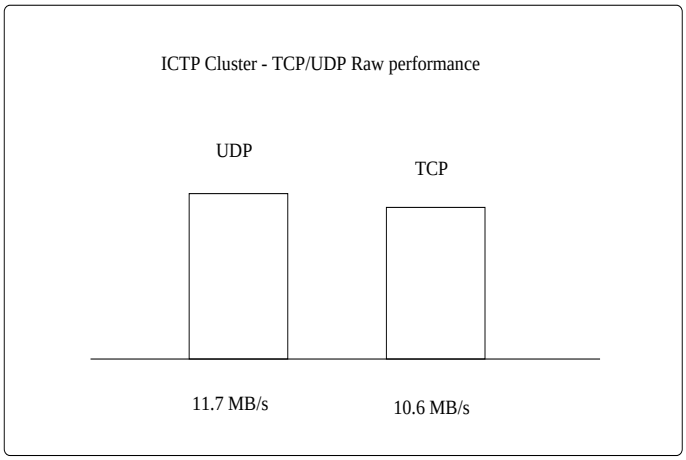

Figure 3: UDP/TCP socket level bandwidth

correction, de-multiplexing and authentication. A preliminary work has been done in a master's thesis by D.Brigthwell some years ago, however this has not generated a complete implementation.

4.2 Using TCP

An implementation of the MPICH's ADI directly over TCP is planned for the near future. In the meantime the so called *ch_p4* device is used. This is an implementation of the ADI through Chameleon/P4. What is unfortunate with TCP is that it comes with some congestion control/avoidance mechanism that while essential on overcrowded WANs, are a mess with high speed networking on LANs/SANs *(See Fig. 4)*. We have found that we can easily have a quite stable

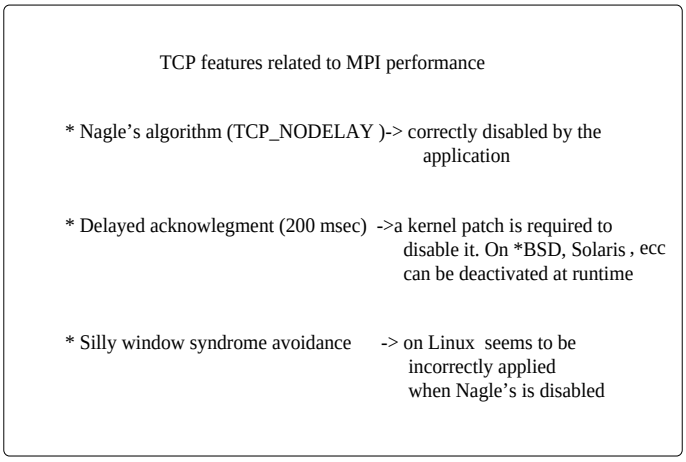

Figure 4: TCP features related to MPI performance

and predictable performance on FreeBSD disabling the delayed ack algorithm (it can be done with *syscontrol*). Linux has a variable delayed ack timeout, this is the reason why it can be difficult to recognize its effects. We can see ordinary 200 msec timeouts as on Berkeley derivatives or *quickacks* with 20 msec timeouts. This 20 msec timeout is for example applied when receiving tiny-grams (segments less in size than half a mss and with the push bit on). The delayed ack algorithm is responsible of 20 msec delays when many small messages are sent only in one direction between 2 nodes. In this case the receiver is delaying his ack in the hope to piggyback the ack to a packet in the other direction *(See Fig. 5)*. On Linux to disable this algorithm it is necessary to

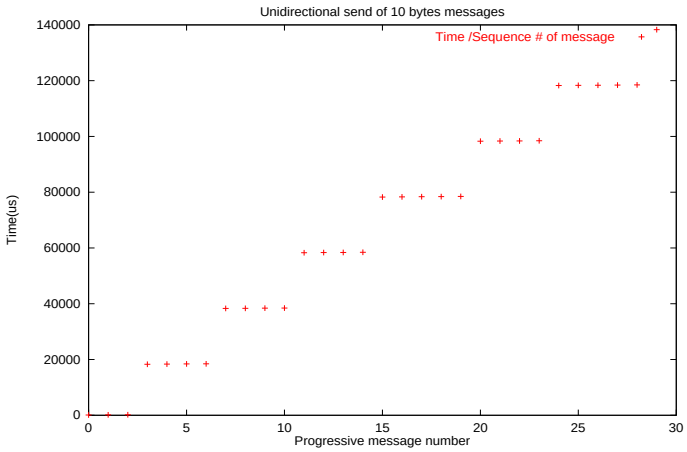

Figure 5: Effect of delayed acks on unidirectional sending

patch and recompile the kernel. We need to recall that the *delayed ack* algorithm is required by the RFCs and so if you disable it your TCP/IP stack should'nt be used on the global Internet.

5 MPICH

MPICH uses different protocols to try to optimize different communication parameters.

5.1 MPICH Internal protocols

There is little relation between these internal protocols and the user level blocking/non-blocking MPI taxonomy. These internal protocols try to solve the buffering problem without penalizing too much the latency of small messages. The protocols are called: *eager*, *rendezvous* and *get (See Fig. 6)*.

5.1.1 - Eager

In the *eager protocol* as soon as a message is posted, the envelope and the data are sent to the receiver. This requires buffering of the unexpected message

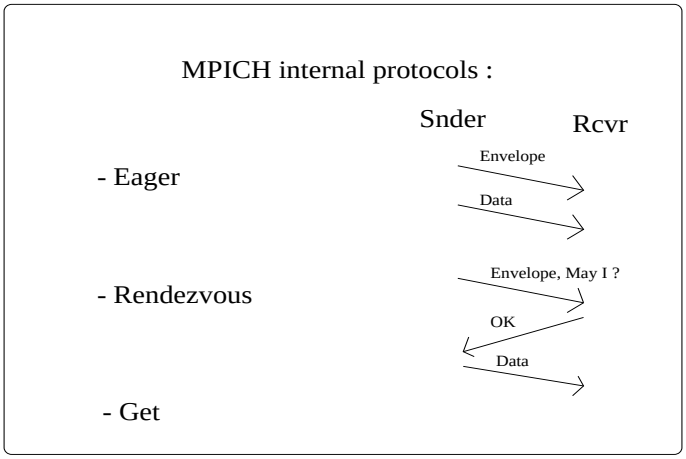

Figure 6: MPICH Internal Protocols

on the receiver side if the receive operation is not yet started and can require an additional copy of the data. This protocol tries to decrease latency and is usually used for small messages (a small variation of *eager* called *short* is used when the envelope and the data all fit in one packet).

5.1.2 - Rendezvous

In the *rendezvous* protocol when a message is posted the envelope is sent to the receiver and eventually buffered there. When the receive is posted and the appropriate envelope has already been received, the receiver sends an acknowledge to the sender. Only after having received the acknowledge the sender sends the data. This protocols requires the receiver to eventually buffer only the envelopes. As it synchronize the receiver, it can avoid an additional copy of the data. It is usually used for large messages.

5.1.3 - Get

The *get* protocol is used by shared memory implementations or when there is special hardware support for remote memory operations. In this case the receiver gets the message usually via a *memcpy* operation. We will not mention it anymore.

5.2 MPICH Performance

Point-to-point bandwidth and bisection bandwidth performance are measured using a *pingpong* test and then dividing by 2 the round trip time obtained.

5.2.1 Point-to-point performance

We have found that the MPICH performance for small messages between 2 nodes (using the *eager* protocol) can be approximated with a linear model having a latency of 104 usec and a bandwidth of 5.58 MB/s *(See Fig. 7)*. With large

messages (using the *rendezvous* protocol) the performance can be approximated with a linear model having a latency of 5.23 msec and a transfer rate of 10.6 MB/s *(See Fig. 8)*. The crossover between the 2 linear models is at about

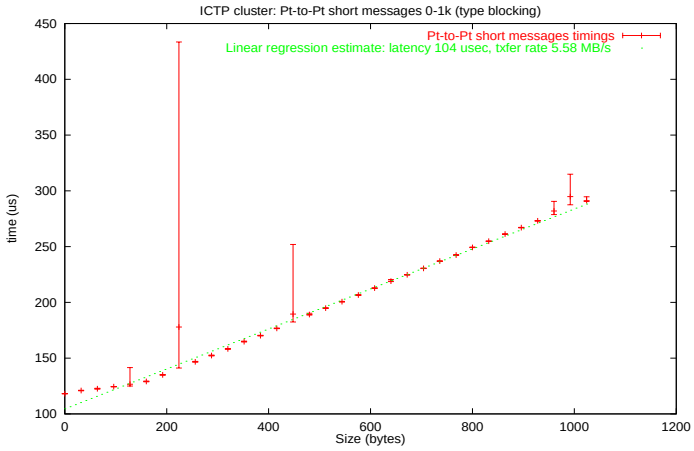

Figure 7: Short messages Pt-to-Pt timings

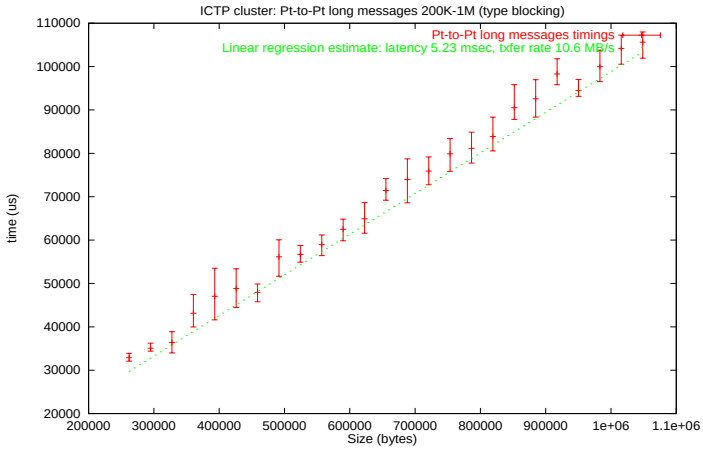

Figure 8: Long messages Pt-to-Pt timings

64 KB where we need to switch between the 2 protocols. This can be specified using an option during the compilation of MPICH.

5.2.2 Bisection bandwidth

Bisection bandwidth tests are done by creating a topology of N/2 pairs communicating simultaneously. Then the average of times over the N/2 pairs is taken. These tests stress the communication network.
For short messages (using the *eager* protocol) we obtain for 8 nodes :

latency 107 usec, bandwidth 5.64 MB/s
and for 16 nodes :
latency 108 usec, bandwidth 5.70 MB/s
that are essentially the figures we obtained for the Pt-to-Pt case *(See Fig. 9/ 10)*.

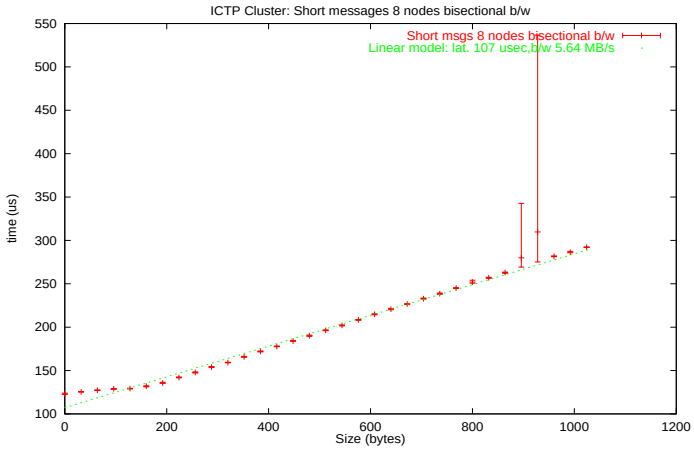

Figure 9: Short messages 8 processors bisection b/w

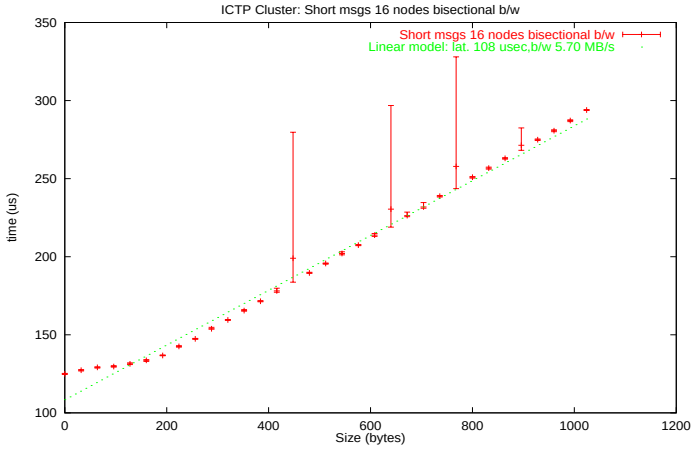

Figure 10: Short messages 16 processors bisection b/w

For long messages (using the *rendezvous* protocol) we obtain for 8 nodes :
latency 5.6 msec, bandwidth 10.52 MB/s
and for 16 nodes :
latency 3.84 msec, bandwidth 9.94 MB/s
again essentially the figures of the Pt-to-Pt case *(See Fig. 11/ 12)*. We can conclude that the switch is non-blocking up to at least 16 nodes.

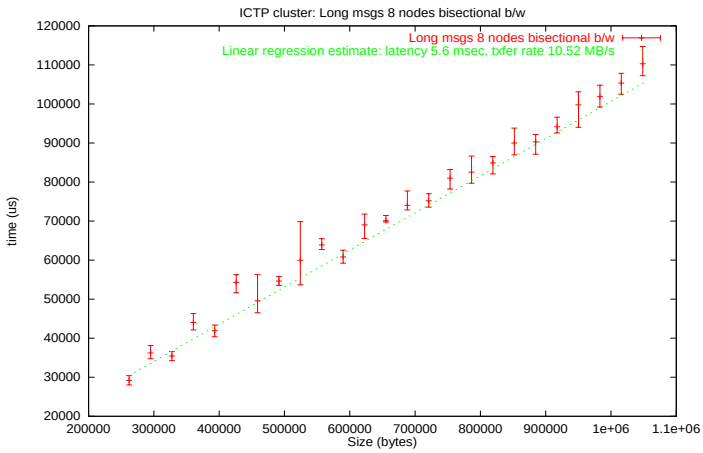

Figure 11: Long messages 8 processors bisection b/w

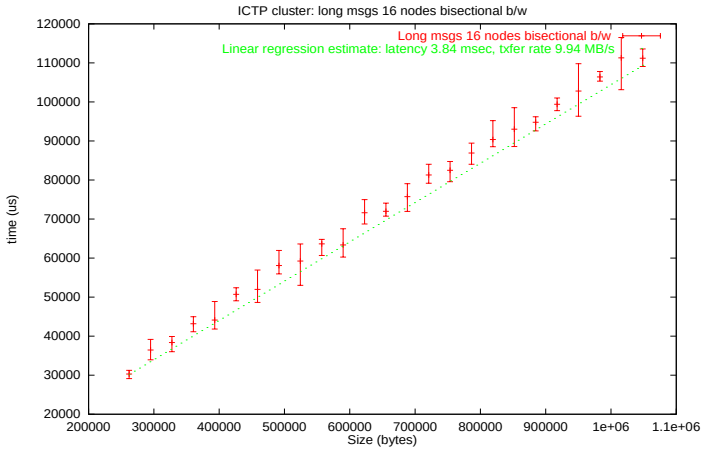

Figure 12: Long messages 16 processors bisection b/w

5.3 Broadcast/Reduce algorithms

MPICH can use a broadcast tree algorithm to implement collective communications.

The number of leaves used by each node can be controlled during compilation. So the algorithm can perform like a linear algorithm (one node sends sequentially to all other nodes/one node receives sequentially form all other nodes) if the number of leaves is set to a number greater than the number of processors (this is the right solution if the processors are interconnected through a hub to avoid collisions), or like a tree of height $log_2 N$ where the root sends to the process N/2 away, and the root and the receiver become each root of a subtree of size N/2 and send to the processor N/4 away and so on. *(See Fig. 13)*. The latter is the right solution if the nodes are interconnected through a

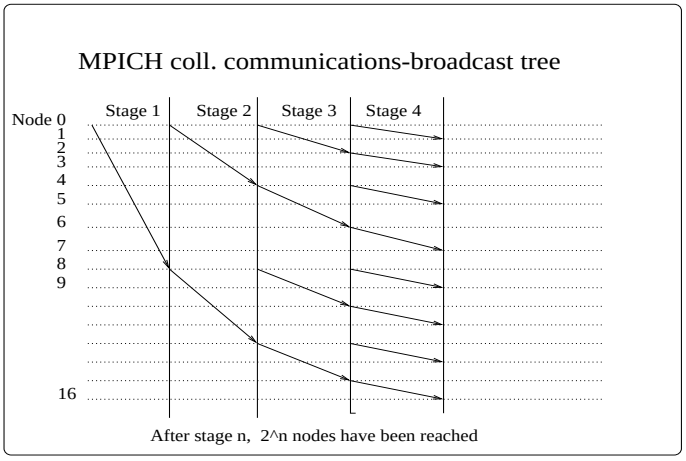

Figure 13: Collective communications broadcast tree

full Fast Ethernet switch like in our case. At stage 4 this algorithm requires an aggregate bandwidth of 8 Pt-to-Pt channels. In this case we can expect times that are $log_2 N$ more than Pt-to-Pt communications times. Unfortunately as the communications in this case are essentially unidirectional, if the delayed ack algorithm is not disabled, the performance of repeated broadcasts can be very poor.

6 Conclusions

We have seen how the performance of MPICH/ch_p4 depends on configurable parameters. We have also seen that the performance depends on features of TCP that are required by the RFCs. We will use Linux because of software availability issues. We have not generally disabled delayed acks, but we have prepared a patched version of the kernel with them disabled. We feel that for the time being, delayed acks can be disabled on a computing cluster if access from/to the internet at large is through an RFC compliant application gateway. An implementation of the MPICH's ADI over UDP would solve the problem of the TCP stalls, but would not significantly decrease the latency that is mainly due to the kernel intervention.

Contents

List of Figures