Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

A Comparison of Three MPI Implementations

Brian VINTER[1]
University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark
John M BJØRNDALEN[1], Otto J ANSHUS and Tore LARSEN
University of Tromsø, N-903700 Tromsø, Norway

Abstract. Various implementations of MPI are becoming available as MPI is slowly emerging as the standard API for parallel programming on most platforms. The open source implementations LAM-MPI and MPICH are the most widely used, while commercial implementations are usually tied to special hardware platforms. This paper compares these two open-source MPI-implementations to one of the commercially available implementations, MESH-MPI from MESH-Technologies. We find that the commercial implementation is significantly faster than the open-source implementations, though LAM-MPI does come out on top in some benchmarks.

1. Introduction

MPI is becoming synonymous with parallel programming, and while far more powerful and advanced models such as HPF[1], JCSP.net[2] and PastSet[3] exist, the performance of MPI implementations are crucial to the overall application performance on clusters. Since most applications are based on MPI, several commercial applications are available only in MPI versions. A search for "MPI programming" on Amazon.com yielded 593 books on the subject. A custom MPI implementation is always delivered with dedicated supercomputers, but for cluster-computers, home-built and brand-name systems alike, one must choose from a variety of MPI implementations where MPICH[4] and LAM-MPI[5] are the best known since they are open-source distributions.

Several commercial MPI implementations are also surfacing, where the best known products are ScaMPI from SCALI[6] and MPI/Pro from MPI-Softtech Solutions[7]. The commercial products often claim improved performance over MPICH and a high level of support as the primary reasons why one needs a commercial MPI implementation[6][7].

One newcomer on the commercial MPI scene is MESH-MPI from MESH-Technologies[8]. MESH-MPI is promoted purely on improved performance and is thus an obvious candidate for a comparative study against the open-source implementations. This study is a quantitative performance comparison that determines if and where MESH-MPI improves the performance over MPICH and LAM-MPI.

This paper is organized as follows. Section 2 introduces MPI and section 3 describes the experiment environment: the cluster and various MPI implementations. In section 4, the benchmarks are presented and, in section 5, the experiment results are presented and analyzed. Finally, we draw our conclusions in section 6.

[1] This author is also associated with MESH-Technologies

2. Message Passing Interface

The Message Passing Interface, MPI[12], is a controlled API standard for programming a wide array of parallel architectures. Though MPI was originally intended for classic distributed memory architectures, it is used on various architectures from networks of PCs via large shared memory systems, such as the SGI Origin 2000, to massive parallel architectures, such as Cray T3D and Intel paragon. The complete MPI API offers 186 operations, which makes this is a rather complex programming API. However, most MPI applications use only six to ten of the available operations.

MPI is intended for the *Single Program Multiple Data* (SPMD) programming paradigm – all nodes run the same application-code. The SPMD paradigm is efficient and easy to use for a large set of scientific applications with a regular execution pattern. Other, less regular, applications are far less suited to this paradigm and implementation in MPI is tedious.

MPI's point-to-point communication comes in four shapes: standard, ready, synchronous and buffered. A *standard-send* operation does not return until the send buffer has been copied, either to another buffer below the MPI layer or to the network interface, (NIC). The *ready-send* operations are not initiated until the addressed process has initiated a corresponding receive-operation. The *synchronous* call sends the message, but does not return until the receiver has initiated a read of the message. The fourth model, the *buffered* send, copies the message to a buffer in the MPI-layer and then allows the application to continue. Each of the four models also comes in *asynchronous* (in MPI called *non-blocking)* modes. The non-blocking calls return immediately, and it is the programmer's responsibility to check that the send has completed before overwriting the buffer. Likewise a non-blocking receive exist, which returns immediately and the programmer needs to ensure that the receive operation has finished before using the data.

MPI supports both group broadcasting and global reductions. Being SPMD, all nodes have to meet at a group operation, i.e. a broadcast operation blocks until all the processes in the context have issued the broadcast operation. This is important because it turns all group-operations into synchronization points in the application. The MPI API also supports scatter-gather for easy exchange of large data-structures and virtual architecture topologies, which allow source-code compatible MPI applications to execute efficiently across different platforms.

3. Experiment Environment

3.1 Cluster

The cluster comprises 51 Dell Precision Workstation 360s, each with a 3.2GHz Intel P4 Prescott processor, 2GB RAM and a 120GB Serial ATA hard-disk[2]. The nodes are connected using Gigabit Ethernet over two HP Procurve 2848 switches. 32 nodes are connected to the first switch, and 19 nodes to the second switch. The two switches are trunked[3] with 4 copper cables, providing 4Gbit/s bandwidth between the switches, see Figure 1. The nodes are running RedHat Linux 9 with a patched Linux 2.4.26 kernel to support Serial ATA. Hyperthreading is switched on, and Linux is configured for Symmetric Multiprocessor support.

[2] The computers have a motherboard with Intel's 875P chipset. The chipset supports Gigabit Ethernet over Intel's CSA (Communication Streaming Architecture) bus, but Dell's implementation of the motherboards use an Intel 82540EM Gigabit Ethernet controller connected to the PCI bus instead.

[3] Trunking is a method where traffic between two switches is loadbalanced across a set of links in order to provide a higher available bandwidth between the switches.

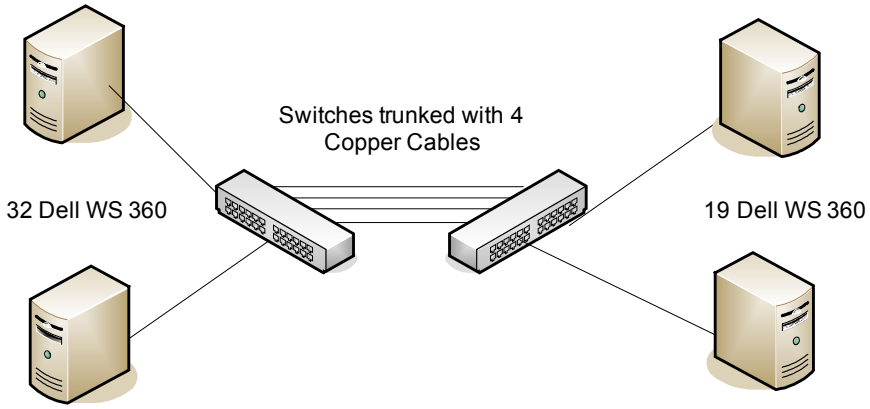

Figure 1: the experiment cluster

3.2 MPICH

MPICH is the official reference implementation of MPI and has a high focus on being portable. MPICH is available for all UNIX flavors and for Windows, a special GRID enabled version, MPICH-G2, is available for Globus[11]. Many of the MPI implementations for specialized hardware, i.e. cluster interconnects, are based on MPICH. MPICH version 1.2.52 is used for the below experiments.

3.3 LAM-MPI

Local Area Multicomputer-MPI, LAM-MPI, started out as an implementation for running MPI applications on LANs. An integrated part of this model was 'on-the-fly' endian-conversion to support different architectures to collaborate on an MPI execution. While endian-conversion still is supported, it is no longer performed per default as it is assumed that most executions will be on homogenous clusters. The experiments in this paper are performed with LAM-MPI 7.0.5.

3.4 MESH-MPI

MESH-MPI is only just released and the presented results are thus brand-new. MESH-MPI is 'yet-another-commercial-MPI', but with a strong focus on performance, rather than simply improved support over the open-source versions. In addition to improved performance, MESH-MPI also promotes true non-blocking operations, thread safety, and scalable collective operations. Future versions have announced support for a special Low Latency Communication library (LLC) and a Runtime Data Dependency Analysis (RDDA) functionality to schedule communication. These functions are not available in the current version which is 1.0a.

4. Benchmarks

This section describes the benchmark suites we have chosen for examining the performance of the three MPI implementations. One suite, Pallas, is a micro-benchmark suite, which gives a lot of information about the performance of the different MPI functions, while the other, NPB, is an application/kernel suite, which describes the application level performance. The NPB suite originates from NASA and is used as the basis for deciding on new systems at NASA. This benchmark tests both the processing power of the system and the communication performance.

4.1 Pallas Benchmark Suite

The Pallas benchmark suite[9] from Pallas GmbH is a suite, which measures the performance of different MPI functions. The performance is measured for individual operations rather than on the application level. The results can thus be used in two ways; either to choose an MPI implementation that performs well for the operations one uses, or to determine which operations performs poorly on the available MPI implementation so that one can avoid them when coding applications. The tests/operations that are run in Pallas are:

- `PingPong`
  The time it takes to pass a message between two processes and back
- `PingPing`
  The time it takes to send a message from one process to another
- `SendRecv`
  The time it takes to send and receive a message in parallel
- `Exchange`
  The time it takes to exchange contents of two buffers
- `Allreduce`
  The time it takes to create a common result, i.e. a global sum
- `Reduce`
  The same as `Allreduce` *but the result is delivered to only one process*
- `Reduce Scatter`
  The same as `Reduce` *but the result is distributed amongst the processes*
- `Allgather`
  The time it takes to collect partial results from all processes and deliver the data to all processes
- `Allgatherv`
  Same as `Allgather`*, except that the partial results need not have the same size*
- `Alltoall`
  The time it takes for all processes to send data to all other processes and receive from all other processes – the data that is sent is unique to each reciever
- `Bcast` - the *time it takes to deliver a message to all processes*

4.2 NAS Parallel Benchmark Suite

The NAS Parallel Benchmark, NPB, suite is described as:

> *The NAS Parallel Benchmarks (NPB) are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. The NPB come in several flavors. NAS solicits performance results for each from all sources.* [10]

NPB is available for threaded, OpenMP and MPI systems and we naturally run the MPI version. NPB is available with five different data-sets, A through D, and W which is for workstations only. We use dataset C since D won't fit on the cluster, and also since C is the most widely reported dataset.

The application kernels in NPB are:

- MG – *Multigrid*
- CG – *Conjugate Gradient*
- FT – *Fast Fourier Transform*
- IS – *Integer Sort*
- EP – *Embarrassingly Parallel*
- BT – *Block Tridiagonal*
- SP – *Scalar Pentadiagonal*
- LU – *Lower Upper Gauss-Seidel*

5. Results

In this section we present and analyze the results of running the benchmarks from section 3 on the systems described in section 2. All the Pallas benchmarks are run on 32 CPUs (they run on 2^x sized systems) as are the NPB benchmarks except BT and SP which are run on 36 CPUs (they run on X^2 sized systems).

5.1 Pallas Benchmark Suite

First in the Pallas benchmark is the point to point experiments, the extreme case is the concurrent Send and Recv experiments where MPICH uses more than 12 times longer than MESH-MPI, but otherwise all three are fairly close. MPICH performs worse than the other two and the commercial MESH-MPI loses only on the ping-ping experiment.

The seemingly large differences on ping-pong and ping-ping are not as significant as they may seem since they are the result of the interrupt throttling rate on the Intel Ethernet chipsets which – when set at the recommended 8000, discretises latencies in chunks of 125us, thus the difference between 62.5 us and 125us is not as significant as it may seem and would probably be much smaller on other Ethernet chipsets.

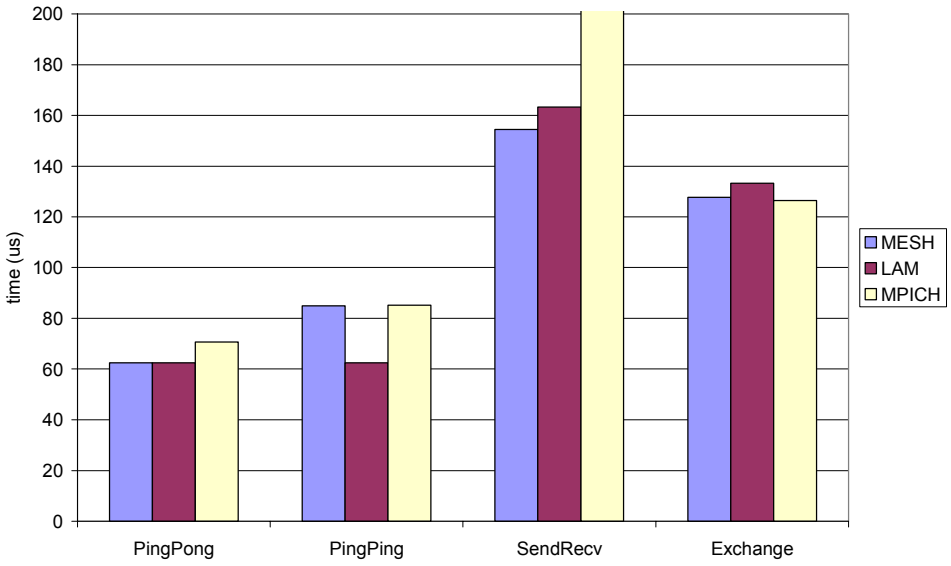

Figure 2: point-to-point latencies from Pallas 0B messages.

Switching to large messages, 4MB, the picture is more uniform and MPICH consistently looses to the other two. LAM-MPI and MESH-MPI are quite close in all these experiments and are running within 2% of each other. The only significant exception is in the ping-ping experiment where LAM-MPI outperforms MESH-MPI with 5%.

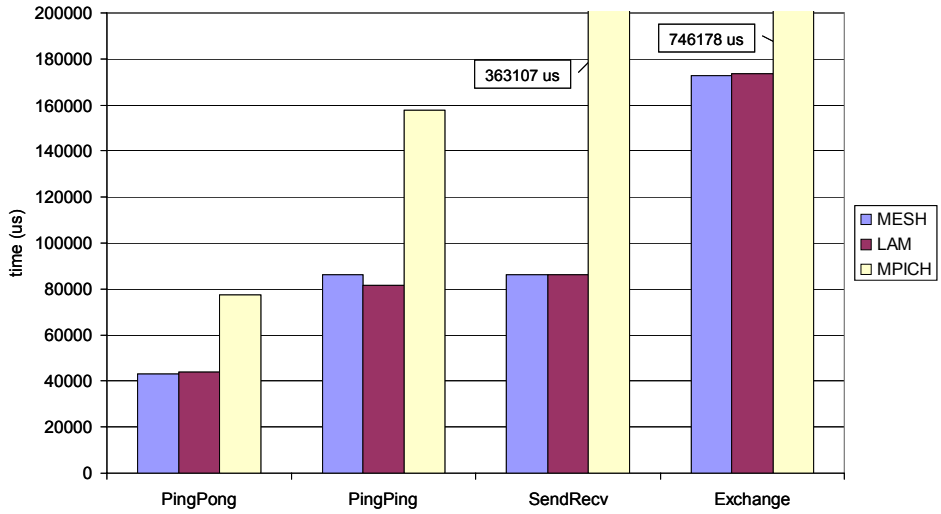

Figure 3: point-to-point latencies from Pallas 4MB messages.

In the collective operations, the small data is tested on 8B (eight bytes) rather than 0B because 0B on group-operations are often not performed at all and resulting times are reported in the 0.05us range, thus to test the performance on small packages we use the size of a double precision number. The results are shown in Figure 4.

In the collective operations, the extreme case is `Allgatherv` using LAM-MPI which reports a whopping 4747us or 11 times longer than when using MESH-MPI. Except for the `Alltoall` benchmark where LAM-MPI is fastest, MESH-MPI is consistently the faster, and for most experiments, the advantage is significant, measured in multiples rather than percentages. The `Bcast` operation, which is a frequently used operation in many applications, shows MESH-MPI to be 7 times faster than MPICH and 12 times faster than LAM-MPI.

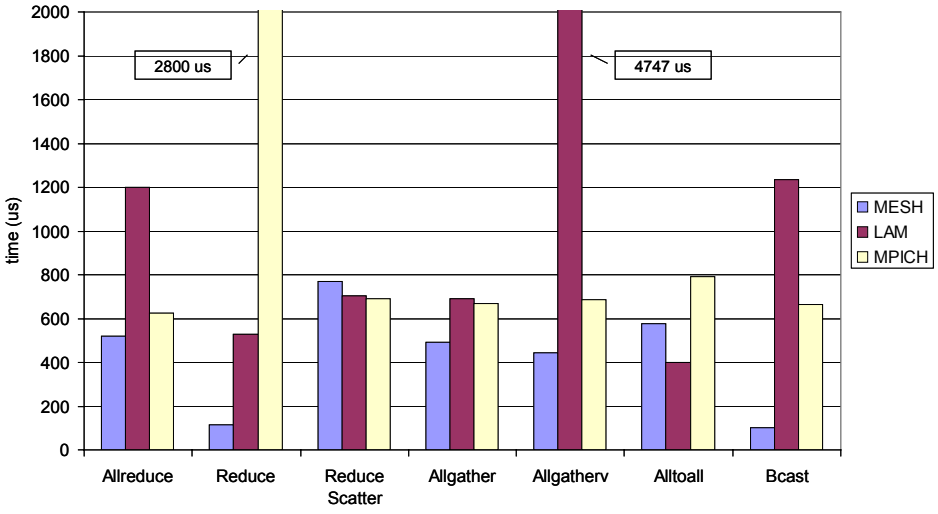

Figure 4: collective Operations latencies from Pallas 8B messages.

For large messages, the results have been placed in two Figures, 5 and 6, in order to fit the time-scale better. With the large messages, MESH-MPI is consistently better than both open-source candidates, ranging from nothing, -1%, to a lot: 11 times. On average MESH-MPI outperforms LAM-MPI with 4.6 times and MPICH with 4.3 times. MESH-MPI is on average 3.5 times faster than the best of the two open-source implementations.

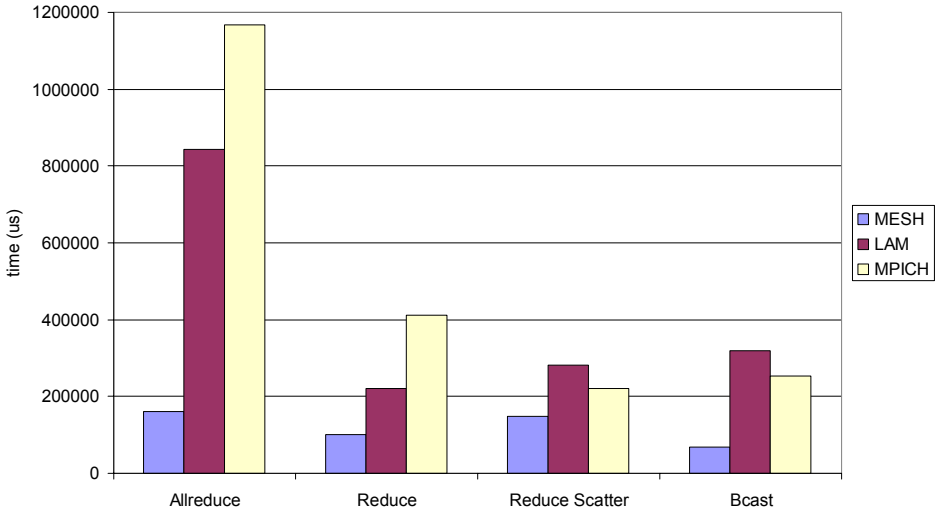

Figure 5: collective Operations latencies from Pallas 4MB messages.

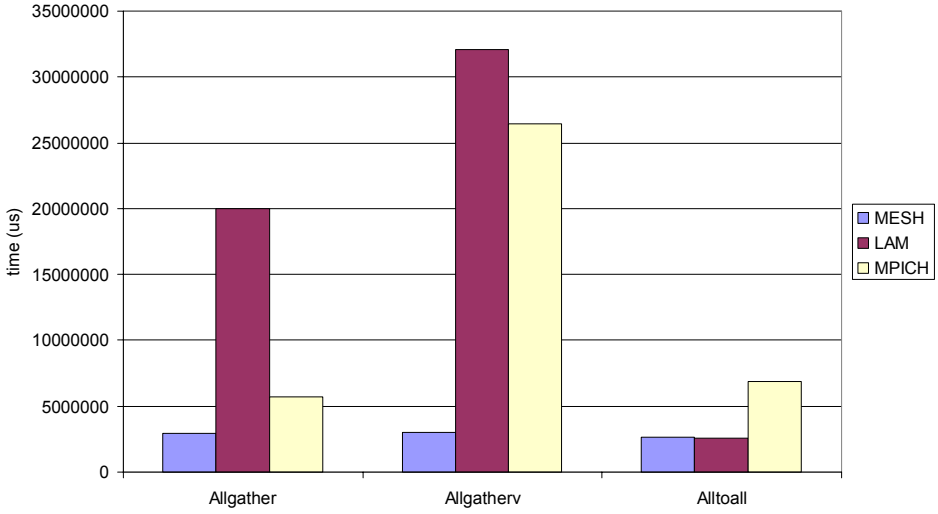

Figure 6: collective Operations latencies from Pallas 4MB messages.

5.2 NPB Benchmark Suite

While micro-benchmarks are interesting from an MPI perspective, users are primarily interested in the performance at application level. Here, according to Amdahl's law, improvements are limited by the fraction of time spent on MPI operations. Thus the runtime of the NPB suite is particularly interesting, since it allows us to predict the value of running a commercial MPI, and it will even allow us to determine if the differences at the operation level performance can be seen at the application level.

The results are in favour of the commercial MPI; MESH-MPI finished the suite 14.5% faster than LAM and 37.1% faster than MPICH. Considering that these are real-world applications doing real work and taking Amdahl's law into consideration, this is significant.

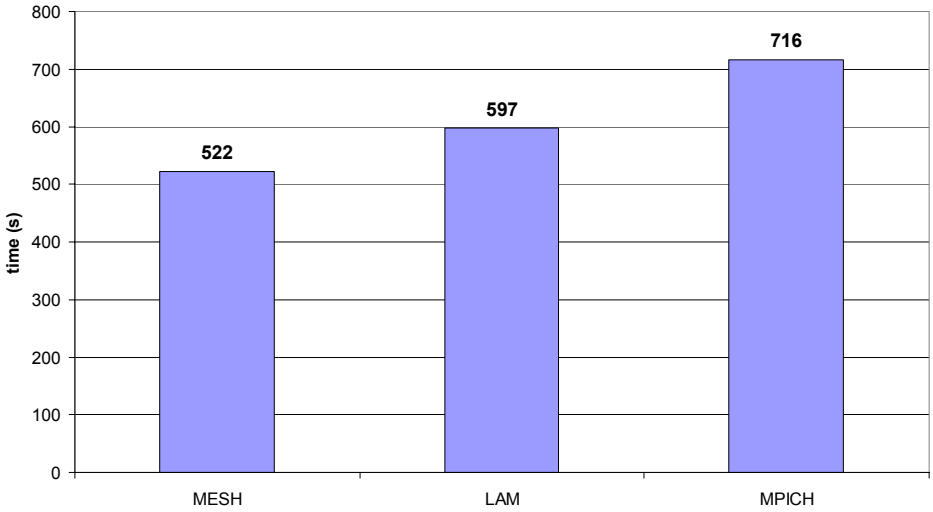

Figure 7: runtime of the NPB benchmark

If we break down the results in the individual applications the picture is a little less obvious and LAM-MPI actually outperforms MESH-MPI on two of the experiments; the FT by 3% and the LU by 6%. Both of these makes extensive use of the `Alltoall` operation where MESH-MPI has the biggest problems keeping up with LAM-MPI in the Pallas tests.

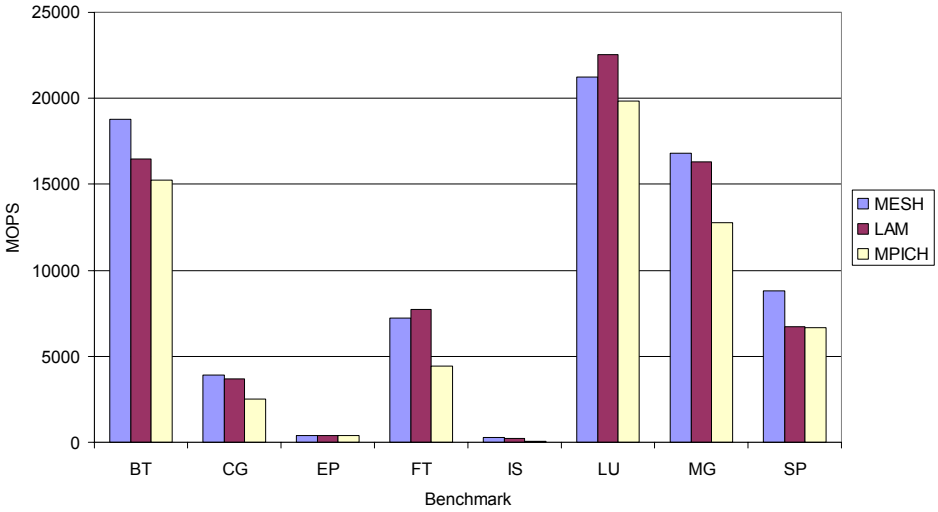

Figure 8: runtime of the individual NPB benchmarks

6. Conclusions

Overall the commercial MPI implementation from MESH-Technologies seems strong and yields results as much as 12 times better than the open source implementations, and at worst 30% worse. The really interesting part is when testing the application level

performance, and a 14.5% improvement over the best open-source alternative is significant. Now, MESH-MPI is a commercial product and the obvious question to ask is what is the value of this MPI implementation compared to other computing costs?

If we look at the experiment cluster used in this work the nodes are priced at € 1428 per node. Thus the intrinsic value of a 14.5% improvement in performance is € 206 and € 530 with the improvement of 37.1% in the case of MPICH. Taking the electricity consumption into consideration with each node using 210W and another 25% for cooling the value rises to € 292 and € 749 in Norway and € 386, respectively € 991 in Denmark.

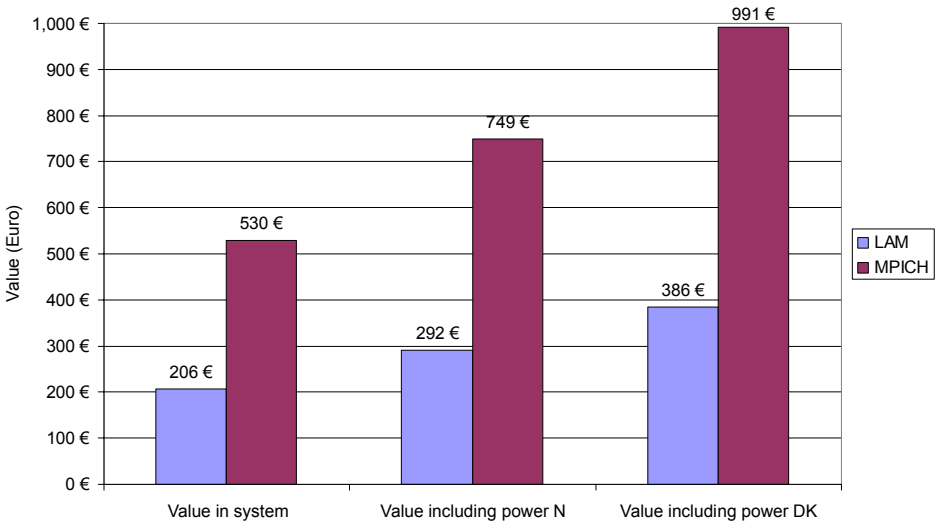

Figure 9: the calculated value of the commercial MESH-MPI per node

Acknowledgements

The authors would like to thank MESH-Technologies for the donation of MESH-MPI for the new cluster installation at the computer science department at Tromsø University. We would also like to thank Morten M. Pedersen for his numerous and detailed comments to this work.

References

[1] An Introduction to High Performance Fortran, John Merlin and Anthony Hey, `ftp://ftp.vcpc.univie.ac.at/vcpc/jhm/jnl/hpf.ps.gz`.

[2] CSP networking for java (JCSP.net). P.H.Welch, J.R.Aldous, and J.Foster. In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, Computational Science - ICCS 2002, volume 2330 of Lecture Notes in Computer Science, pages 695-708. Springer-Verlag, April 2002. *See also* `http://www.quickstone.com/xcsp/jcspnetworkedition`.

[3] Brian Vinter, Otto J. Anshus and Tore Larsen. *PastSet A Distributed Structured Shared Memory System*, in the Proceedings of High Performance Computers and Networking Europe, Amsterdam April 1999.

[4] William Gropp and Ewing Lusk and Nathan Doss and Anthony Skjellum, High-performance, portable implementation of the MPI Message Passing Interface Standard, Parallel Computing", Vol 22, no 6, 789--828, 1996.

[5] Greg Burns and Raja Daoud and James Vaigl, LAM: An Open Cluster Environment for MPI, Proceedings of Supercomputing Symposium, 379--386, 1994.

[6] `hhttp://www.scali.com/`

[7] `http://www.mpi-softtech.com/`

[8] `http://www.meshtechnologies.com`

[9] http://www.pallas.com/e/products/pmb/index.htm
[10] http://www.nas.nasa.gov/Software/NPB/
[11] http://www.globus.org/
[12] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. Parallel Computing, 20(4):657–673, March 1994.