Internal Report 2012-2013-16 August 2013

Universiteit Leiden
Opleiding Informatica

The Leiden Zipper

a bridging Architecture

Florian Treurniet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Clusters in computing grids are usually connected through a WAN. This thesis investigates an alternative way of connecting two clusters which seems suitable for cluster grids and campus grids: the 'Leiden Zipper'. A class of applications is presented which would benefit from such a bridging architecture. Finally a software forwarder designed especially to run MPI on the 'Leiden Zipper' architecture is presented and evaluated on two zipped QDR InfiniBand clusters.

1 Introduction

With the need to share large datasets and to solve problems that are too big to handle on one computing cluster, the need started to rise to share computing power among different institutions across the globe. One of the envisioned ways to do this, is to couple different computing clusters, making the computing resources globally available, much like an electric grid, hence called *grid computing* [4].

The way computers are connected and therefore the way clusters, being collections of computers, are connected has great impact on both the performance and the cost of the aggregated system. For instance, take the fully-connected topology, in which every computer is directly connected to all the other computers individually. This is ideal for performance, because each pair of computers can communicate through a dedicated line. For cost however, it is not, since $n(n-1)/2$ links are needed when communication may go in both ways over one link to connect n computers. This results in hardware costs scaling with $\Theta(n^2)$ which is costly.

At the other side of the spectrum are daisy-chained systems, where each computer is connected to two other computers, creating a chain or a ring. To do this in the chain case, only $n-1$ links are needed for n computers, thus the cost scales by $\Theta(n)$ which is relatively cheap. However, for computers at both ends to talk, $n-2$ computers need to forward the signal, which takes time. This results in high latencies and bad performance. Furthermore, if two computers need to use one of the shared cables, packets might have to wait for other transfers to finish, resulting in higher transfer times. These are some of the reasons why modern connection topologies are compromises between cost and performance. Tree structures, like the Fat trees [13], are popular connection topologies. In these topologies, the amount of forwards, or hops, scale by $\Theta(\log(n))$ and the hardware costs scale by $\Theta(n\log(n))$. Also high-dimensional structures like hypercubes and multi dimensional torusses are commonly used.

In grid computing, clusters are usually connected through a WAN, which usually is a collection of the topologies described above. Some even see being connected through a WAN as an essential feature of grid computing [16]. This thesis investigates an alternative way of connecting two clusters, namely by zip-

ping them together using a set of computing nodes which are connected to both clusters at the same time (hence nicknamed 'The Leiden Zipper'). The way this is done is described in Section 2.

In Section 2.2 it is shown that connecting clusters using the Leiden Zipper bridge architecture, in theory has properties which enable certain classes of parallel programs to suffer from less overhead than they would have experienced if they had been connected using classical approaches, that connect different subnets by putting a set of routers in between.

Besides benefits, there are some problems to deal with. Although the 'Leiden Zipper' could be linked in any cluster using any type of interconnect, this thesis will focus on InfiniBand, a popular interconnect for high performance computing clusters. Some of its features will be described in Section 1.1. InfiniBand only allows the routing of packets from one subnet to another by a router [1]. Since at the time of writing no InfiniBand-to-InfiniBand router or bridge was available that could connect InfiniBand networks to each other that both use copper cables (only InfiniBand to Fibre or InfiniBand to Ethernet), a different solution was sought and found in the form of a forwarding process (the forwarder) to enable communication between clusters.

Since MPI is a de-facto standard in the field of high performance computing, some functions of it were implemented to see how a forwarding process would influence the performance of several applications. This implementation, called MMPI, consists of a layer on top of Open MPI [7] which decides if communication will go through the forwarder or stays in the local cluster. MMPI including the forwarder is further described in Section 3. The performance of the forwarder is also evaluated in Section 3 using a subset of the OSU benchmarks. In Section 4 the performance of MMPI on a subset of the popular NAS Parallel Benchmark (NPB) is analysed.

1.1 Introduction to InfiniBand

InfiniBand is a popular Storage Area Network interconnect for high performance computing clusters because it allows for high bandwidth and low latencies. This high performance is achieved by offloading as much of the protocol (the first 4 layers in the OSI model) as possible to the hardware and allowing the hardware to access memory directly. In addition to that, InfiniBand allows for user-space access to the hardware. In this way the operating system can be bypassed as much as possible, saving CPU cycles for computing and allowing InfiniBand to be a zero-copy protocol. InfiniBand is also a lossless protocol, meaning that packets are never dropped, however this does mean that the appropriate buffers must be allocated before transfer starts.

The main building blocks of InfiniBand are Host Channel Adapters (HCAs), Target Channel Adapters (TCAs), switches, routers and range extenders. InfiniBand uses a switched fabric topology, which means that all compute nodes and

I/O devices lie on the edge of a network created from switches, routers and range extenders. So all communications over the fabric either start or end at a channel adapter.

Channel adapters come in two forms. TCAs are used to connect I/O devices to the InfiniBand fabric. HCAs are used to connect the compute nodes to the InfiniBand fabric and contain all the hardware needed to allow the user to control the fabric. Software access to the switched fabric is defined by a minimum set of functionalities called Verbs, which is the native way to send information over InfiniBand. Data can either be sent using send/receive operations, or by remote direct memory access (RDMA). The first one makes use of sending buffers. The latter allows the user to write directly into the memory of the remote machine. InfiniBand supports both connectionless and connected transfers, which both may be reliable or unreliable.

InfiniBand switches are used to connect channel adapters within a subnet to each other. An InfiniBand subnet is a part of the switched fabric that is managed by a subnet manager. The subnet manager is a central identity that monitors the subnet, assigns addresses within the subnet and sets the routing tables within the switches. The addressing within a subnet is done with a Local IDentifier (LID). The switches use virtual cut-through switching [11] and flow control is credit based.

Several subnets can be linked together by routers. For routing in between subnets, packets get an IPv6 compliant header. This means that addressing in between subnets is done using a 128bit Global IDentifier (GID). The packet will only receive the global routing header if it is known that the receiver is not present in the local subnet.

As for the cabling, InfiniBand supports both copper and fibre. Copper cables can support ranges up to 30 meters, fibre can support distances up to several kilometers. Cables and hardware allow for different speeds. The InfiniBand 1.2.1 specification describes for SDR (2.5Gb/s), DDR (5Gb/s), QDR (10Gb/s), and the 1.3 specification allows even higher speeds like FDR (14Gb/s) and EDR (26Gb/s). Note that since SDR, DDR and QDR use 8b/10b encoding, only about 4/5th of the speed is achievable. FDR and EDR use 64b/66b encoding to allow for higher efficiency.

InfiniBand is full duplex since it has a dedicated lane for each direction. Also, higher bandwidth can be achieved by 4x, 8x and 12x cables, which pack the above mentioned 1x cables in bundles of respectively 4, 8 and 12 [1].

Common topologies for InfiniBand clusters include the fat tree, hypercube and torus topologies.

2 The Leiden Zipper

2.1 Architecture

The idea behind the 'Leiden Zipper' is to connect two clusters by providing a set of 'zipper' nodes. These nodes are connected to the network of both computing clusters without letting those networks be physically connected to each other. This is depicted in Figure 1.

This schematical representation suggests a way of connecting more than two clusters using the zipper nodes. When looking at the zipper nodes as the parent, and the clusters as children of this parent a tree structure seems logical. It might be possible to again connect this zipped system to another zipped system with another set of zipper nodes, forming a tree. This could give rise to a scalable solution where the amount of zipper structures when connecting n clusters scales with an order of $\Theta(n \log(n))$. The average amount of zipper nodes which have to forward data when communicating between two clusters could scale by $\Theta(\log(n))$.

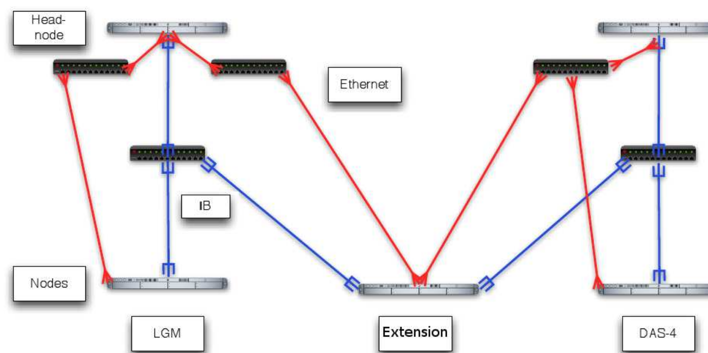

Figure 1: A schematical representation of how the zipper nodes (Extension) are connected to the existing clusters. The red lines are the Ethernet connections, the blue lines the InfiniBand connections. The black boxes are the network switches and the grey boxes are compute/head nodes.

The fact that both networks are physically disconnected causes some issues with InfiniBand. A process in one cluster might need to communicate with a process in another cluster. This requires the routing of InfiniBand packets between clusters. Routing InfiniBand packets from one subnet to another requires an InfiniBand Router or a dual port Host Channel Adapter (HCA) with routing capabilities [1]. At the time of writing, it is possible to get a router to route InfiniBand packets from copper InfiniBand cables to either fibre or Ethernet, but no routing between copper cabled subnets is possible. So other solutions are needed.

The other possibility is to use a software forwarder, but where IP forwarding is a common Linux functionality, found in its network stack [15], InfiniBand for-

warding is not. There are several reasons why this is not common practice, but the most important one is described in Section 1.1: all communication in InfiniBand networks starts and ends at a channel adapter. This is a direct consequence of the protocol, which does not allow delivery of a packet to any channel adapter other than the receiver. But even if you would be able to let the subnet manager think that the HCA is in fact a router, you would have to do the interpretation of the IPv6 header on the CPU (since it is not hardware supported). This would nullify one of the features of InfiniBand which makes it so beneficial: offloading the protocol to the hardware.

However, solutions like RDMA over Ethernet are possible which send InfiniBand traffic over Ethernet with relatively good performance [18]. However, if both clusters use 4x QDR (40Gb/s), a 10Gb Ethernet connection in between them will be quite a bandwidth reduction and thus a bottleneck.

In an attempt to get a better performance than by connecting the clusters using RDMA over Ethernet, a software forwarder is used. To minimize the overhead of calculating the inter-subnet routing on the CPU, MPI ranks are used for addressing, which is simpler than the IPv6 protocol. The full solution is described in Section 3.

2.2 Fully utilizing the zipper architecture

The best performance can of course be obtained when links with relatively low bandwidth can be avoided. In clusters that are not zipped together, but are connected by a high-latency, low-bandwidth WAN, only applications which have partial calculations that can be fully contained in a cluster can avoid those links. However, zipped clusters in theory support more types of applications.

Let Γ be the communication graph of application A. The vertices of Γ are processes. Two vertices in Γ are connected by an edge if and only if at some time during the computation of A there is inter-process communication between the corresponding processes.

Applications that have a communication graph Γ that contain a vertex-cut set C, a minimal set of vertices that, when removed, disconnect Γ in at least two graphs, can in theory communicate without overhead by issuing the processes corresponding to the vertices in C to the zipper nodes. Of course, there should be enough cores in the zipper nodes to reasonably host these processes.

Other communication graphs Γ that represent applications that can be run theoretically without overhead, are ones that have a cut-set ϵ, a minimal set of edges that, when removed, partition Γ in at least two graphs: Ω and Σ. The processes that should in this case be issued on the zipper nodes for maximum performance, either correspond to all the vertices in Ω that are connected to the edges in ϵ, or correspond to all the vertices in Σ that are connected to the edges in ϵ. Again, the zipper nodes must have enough cores to be reasonably able to host these processes.

Of course, to be able to fully use this possibility, the zipper nodes must be zipped into both clusters using a high performance interconnect. This means that the 'Leiden Zipper' architecture can best be used to create *cluster grids* and *campus grids* which, according to [16], are constructed from clusters that are geographically close to each other.

2.3 An overview of the zipper architecture

This section describes two real clusters zipped together by zipper nodes. This is also the testbed for all benchmarks.

The two clusters that are zipped together are the Little Green Machine (LGM) cluster and the cluster consisting of the nodes of the Distributed ASCI Supercomputer 4 (DAS4) that are located in Leiden. These were connected by 4 zipper nodes. Schematical views of the processor environment in the computing nodes of the LGM, the DAS4 and the zipper nodes presented in this section were made using the `lstopo` tool in the *hwloc* framwork [3].

The LGM and DAS4 nodes used in the experiment have a 2.40GHz Intel Xeon E5620 processor, consisting of 2 sockets with each 4 cores with 4 more made available by HyperThreading. The cache hierarchy can be found in Figure 2. They also use the Mellanox MT26428 HCA adapter, which is hooked to the machine throug PCI express 2.0, to connect to the InfiniBand fabric, allowing for a 4xQDR speed. Both clusters have a separate crossbar switch to which all nodes of a cluster are connected. The LGM uses an IS5030 switch, the DAS4 is connected through an IS5025 switch. Both systems are also connected through 10GigE, which is not used during experiments except for starting up the processes.

There are some differences between the machines. The LGM nodes have 24GB RAM and the DAS4 nodes have 48GB RAM. Besides that, the LGM nodes also contain a graphics card, but it is not used during the experiments. More important is the fact that the DAS4 nodes run CentOS6 kernel 2.6.32-358.14.1.el6.x86_64, while the LGM nodes run CentOS5, kernel 2.6.18-348.6.1.el5. Since no binary compatibility is guaranteed between the two releases, the benchmark software was compiled separately for both systems. As for the InfiniBand drivers, the DAS4 uses the libmlx4-1.0.4-1.el6.x86_64 userspace drivers, LGM and the zipper nodes use libmlx4-1.0.2-1.el5.

The zipper nodes are a bit different. They contain 63GB of main memory and a 2.0GHz Intel Xeon E5-2650 processor with 2 sockets containing 8 cores per socket and an extra 8 per socket through HyperThreading. The cache hierarchy is shown in Figure 3. The zipper node contains a dual-port HCA, the Mellanox MT27500. One port of it is connected to the central crossbar switch of the DAS4 and one is connected to the crossbar switch of the LGM. The zipper nodes use the same software and operating system as the LGM.

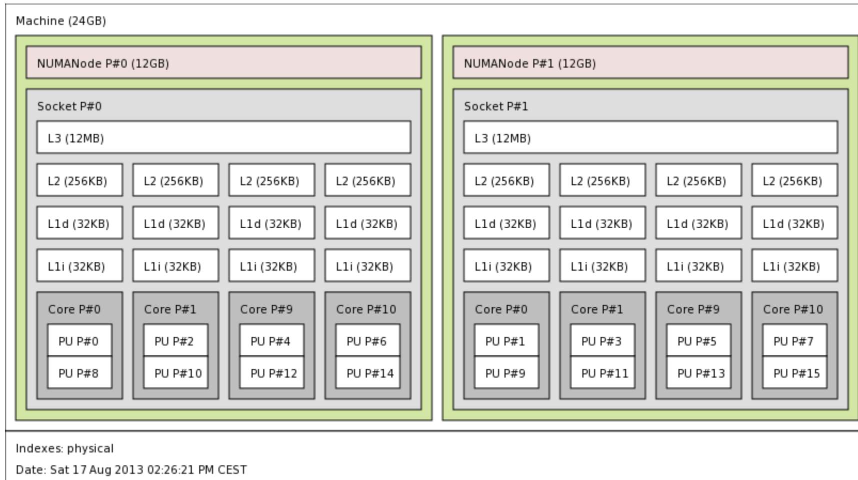

Figure 2: The system topography of a LGM node, which has the same memory hierarchy as a DAS4 node.

As for the software, two binary versions are used. The benchmark software that runs on the DAS4 is compiled using mpicc which wraps gcc 4.4.6. The LGM and zipper nodes use software compiled with gcc 4.1.2. All software was compiled with the `-O` optimization flag. The MPI library that is used is OpenMPI 1.4.4.

It must be noted that DAS4 has a reservation system, but the LGM has not. During experiments on the LGM, several incidents were noted of users staying logged in on nodes and occassionally running programs on them, which might have interfered with the benchmarks running on those nodes.

3 MMPI

MMPI is a layer on top of Open MPI which allows communication between two clusters to go through a forwarding process, while using the 'native' Open MPI functions to communicate within a cluster. It must be noted that it is not a real MPI implementation since no real effort has been made to be conform the MPI specification. Applications using MMPI must be slightly rewritten: a seperate header file must be included and an M must be placed in front of MPI calls from which the user wants to use the MMPI version. This is because of technical reasons, and not because of a deliberate breaking with the MPI semantics. Also application startup is made a more complex, and must be done using a shellscript, because a forwarding process has to be started on a zipper node.

The forwarding process is a separate process to which other groups of processes can connect using the `MPI_Lookup_name`, `MPI_Connect` and the `MPI_Accept` functions. These dynamic process management functions are described in the MPI-2

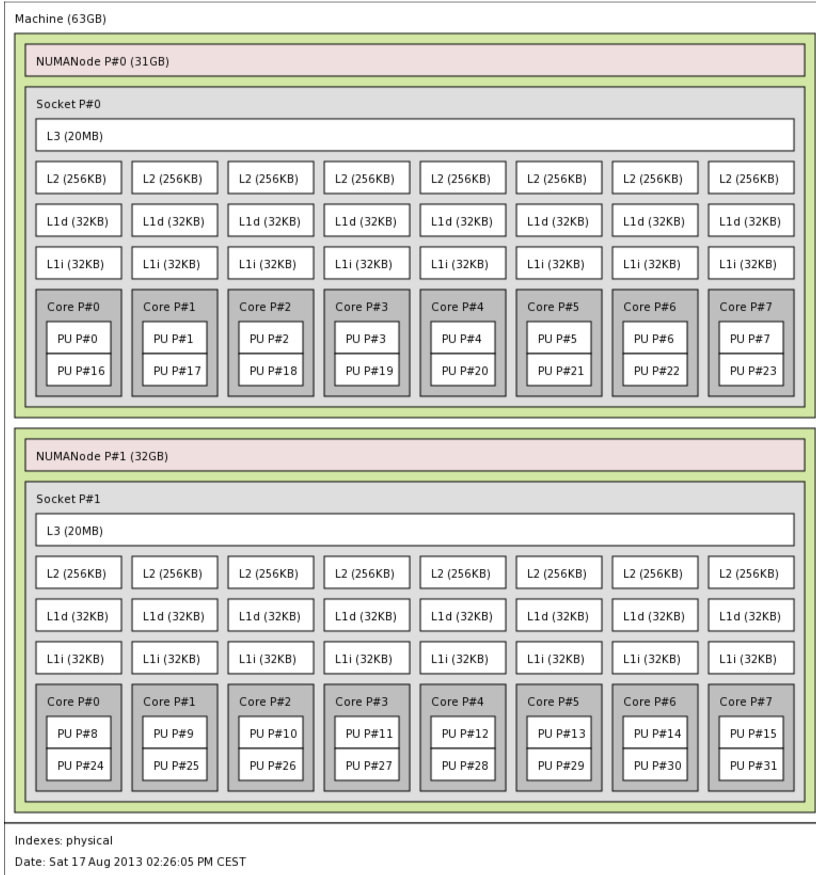

Figure 3: The system topography of a zippernode.

standard [9]. The main arguments that supported implementing the forwarder using MPI functions, and not using the more low-level InfiniBand Verbs API were code complexity, performance and portability. With the MPI functions used for communication, any MPI library supporting MPI-2 features can be used to make MMPI run. Because of that, depending on the MPI library used, more interconnection networks than only InfiniBand can be supported. Also, most MPI libraries are highly optimized, which will benefit the performance of MMPI.

After the connection, the forwarding process will have rank 0 in both communicator groups. This means there are double rank numbers in both groups of processes. To continue to be able to use ranks to uniquely address a process, virtual ranks have been implemented similar to the method described in [8].

Basically one of the two groups is chosen to be the 'low' group, and the other group will be the 'high' group. Let us say that the 'low' group has x processes, not including the forwarder, and the 'high' group has y processes. Now processes 1 till x will get virtual rank 0 till $x-1$. The processes 1 till y in the high group will receive virtual rank x till $x+y-1$.

3.1 OSU MPI Benchmarks

The next few subsections will describe the implemented MPI functions and show their performance compared to the Open MPI implementation. The benchmarks used consist of a subset of the OSU MPI Benchmarks 4.0.1 (OMB). These are the classical microbenchmarks for evaluating MPI performance. Both point-to-point and collective benchmarks were used. From the point-to-point benchmarks, `osu_bw`, `osu_bibw` and `osu_latency` were used. The first of these, `osu_bw` measures the average sustainable bandwidth by issueing `MPI_Isend` 64 times after each other, from process 0 to process 1, and then waits for the completion of these sends. It then waits for an acknowledgement from the other process. This is done 100 times in succession for different transfer sizes.

The `osu_bibw` benchmark measures the bidirectional bandwidth. It works about the same as `osu_bw`, with the difference that both process 0 and 1 send to each other, again using non-blocking `MPI_Isend` and `MPI_Irecv`.

The last of the point-to-point benchmarks, `osu_latency` measures the point-to-point latency using a classical ping-pong benchmark, implemented by blocking sends and receives. The latency is measured by averaging over 10000 of such ping-pong events for each different transfer size.

The reported point-to-point benchmarks are all averaged over 15 runs of the benchmark.

The collective benchmarks measure the average time it takes to execute a certain MPI function. This is done by executing this function a thousand times across a certain amount of processes for different transfer sizes. On each process the execution of these MPI calls is timed, and the times reported are the average of all these times.

The measured transfer sizes lie between 1B and 4MB for the point-to-point benchmarks. The transfer sizes of the collective benchmarks range up to 1MB. This gives us the appropriate tools to evaluate the results reported by the NAS Parallel Benchmarks described in Section 4 since the bulk of the communications made by those benchmarks lie in those ranges [20].

To make a fair comparison of the MMPI version and the MPI version of the applications possible, the processors are laid out on the cores in a specific way during each of the benchmarks.

When running MMPI, the only thing running on the zipper nodes is the forwarding process. This is done to simplify comparison of the performance, excluding factors like overhead of running an extra forwarding process next to the computation processes. When adding extra nodes to the computation, nodes are added in pairs, one on the LGM cluster and one on the DAS4. This means that the number of processes running on LGM is equal to the number of processes running on DAS4.

The node layout for MPI on a single cluster is the same. Processes are added in pairs, but they are added to different nodes in the same cluster, simulating

the same layout of processes on the core, but instead of a forwarder, it just has a normal InfiniBand link in between.

For both MMPI and MPI, when one process of a pair is added to a cluster, it will be added in such a way that as less nodes as possible are in use and as many cores as possible are in use on a single node. The stress on the L3 cache in a machine is distributed as evenly as possible. This is done by dividing the processes evenly among the sockets. Lastly, only physical cores are used, this is done in order to prevent side-effects cause by hyperthreading. The resulting

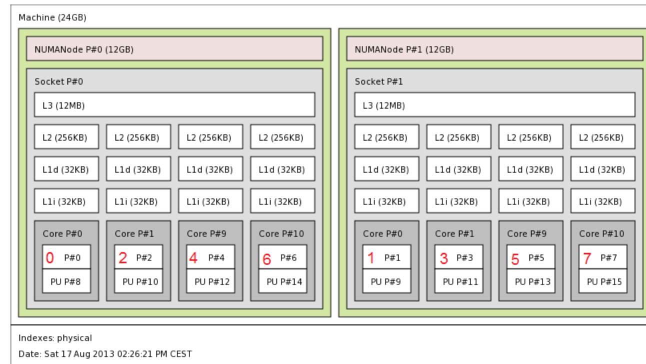

Figure 4: An example displaying how the native MPI ranks (red) are distributed over the cores of a processor. Note that all these processes have a sibling process in the remote cluster, which are laid out in the exact same way on a node in that cluster.

scheme, illustrated in Figure 4 is that in a cluster (LGM), the process with native MPI rank 0 is assigned to core 1 of socket 1, process 1 is assigned to core 1 of socket 2, process 2 is assigned to core 2 of socket 1, etc. This is done till all physical cores of a node are taken, in our case the amount of physical cores is 8. So process 8 is added to the first core of socket 1 of a new machine, etc. Note that because the processes are added in pairs, the layout is copied in the remote cluster (DAS4).

Because of the order of process start-up, during MMPI runs, the LGM always contains the lower-ranked processes. So the process with virtual rank 0 is always located in the LGM cluster.

3.2 Handshaking protocol

Nearly all the implemented MPI operations (except for barrier) use a handshaking protocol to communicate with the forwarder. This results in easier programming because you do not have to split up messages bigger than some size of a pre-allocated buffer. A schematical representation of the handshaking protocol in the case of a point-to-point transfer can be found in Figure 5.

The request is initiated by sending the forwarder information concerning the message type (point-to-point, broadcast, reduce, etc.). Depending on the transfer type, information included might be the payload size, addressing information like the virtual rank of the other process and the message tag. This MPI message has a payload of 7 integers.

The forwarder matches this information with pending transfer requests. If no matching request is found, the forwarder adds this request information to the list and adds an identifier (ID) to the transfer. This ID is send back to the request initiator. A blocking call is used to receive the ID.

If there is a matching request, buffers of sufficient size are allocated on the forwarder to enable the transfer. Then the request ID of the matching request is send back. This ID is used as a tag to identify the real transfer of the data.

At this moment, all parties involved in the communication have the transfer ID. The actual transfer is then handled, after which the forwarder will free the buffers and again starts waiting for new transfer requests.

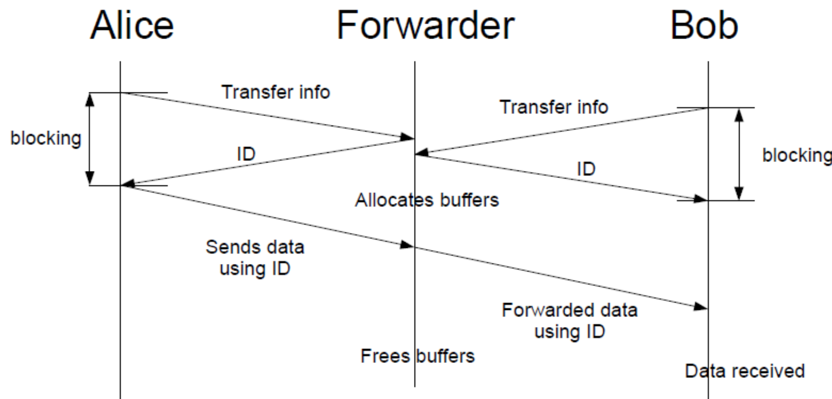

Figure 5: The handshaking as it is used in a point-to-point transfer between Alice and Bob. The part marked by 'blocking' is the actual handshaking. The handshaking is blocking because it is implemented with blocking MPI calls.

3.3 Point-to-point

The MPI point-to-point operations simply deliver a message between two processes. The sender has to call a send function and the receiver has to call a receive function. Both blocking versions of the basic send and receive operations (`MPI_Send` and `MPI_Recv`) and the non-blocking operations (`MPI_Isend` and `MPI_Irecv`) have been implemented.

After the handshaking has succeeded, the sending side of the transfer will transmit the buffer to the forwarder. The forwarder receives this information using a blocking receive call and then forwards it to the receiver.

Note that even the non-blocking operations will block during the handshaking since the reception of the transfer ID is done using a blocking call. The difference is that the non-blocking MMPI functions use non-blocking MPI calls to receive the real data. The performance of the point-to-point operations are evaluated

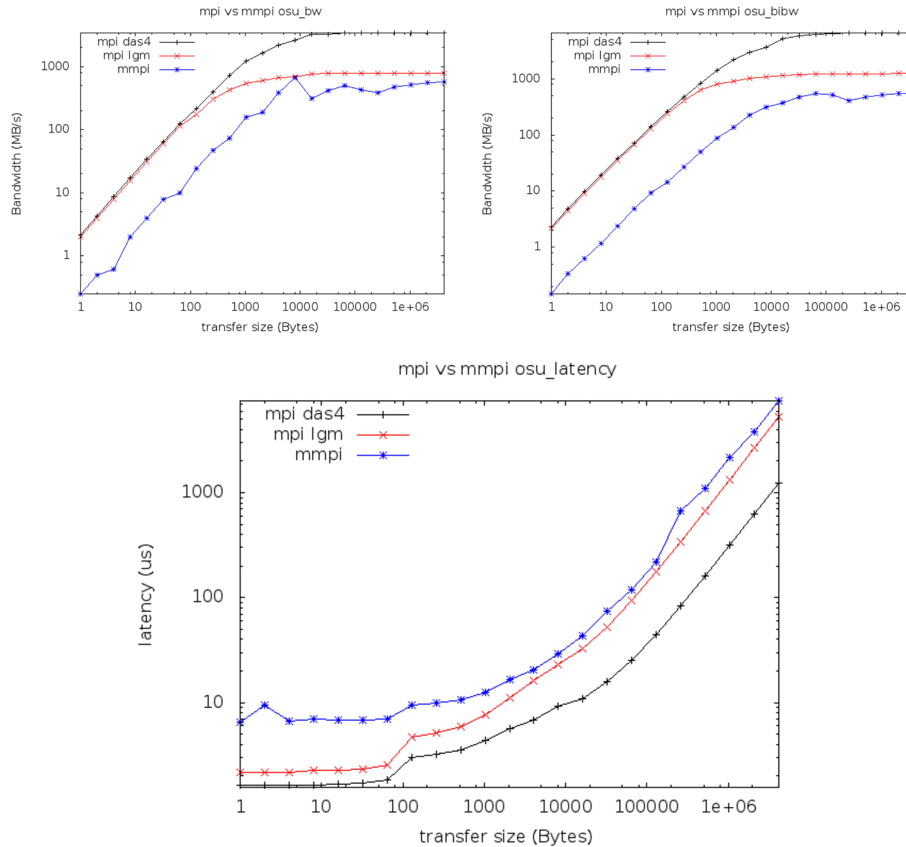

Figure 6: Bandwidth, bi-directional bandwidth and latency results for the MMPI and MPI implementations for the point to point communication

in Figure 6. By looking at the bandwidth plots, several things can be noted. First of all, although having the same HCAs, the LGM and the DAS4 network perform quite different. The DAS4 cluster reaches about 3500MB/s, which is quite similar to bandwidth of the QDR ping-pong benchmarks reported in [14]. Nevertheless, the LGM cluster only reaches 780MB/s, an interesting but yet unexplained difference.

Another, more relevant, fact can be found comparing the bi-directional bandwidth plot with the uni-directional bandwidth plot. InfiniBand is full duplex with separate lines for each direction, so for both the LGM and the DAS4 the bandwidth doubles in the bi-directional case. However, this is not the case for MMPI, which uses the forwarder during communications. Since the forwarder can only

handle one request at a time, it is half duplex. Therefore the maximum bandwidth does not increase when communicating in both directions. This could be solved by either making the forwarder multithreaded, enabling it to handle more transfers at a time, or by creating a separate forwarder for communications in each direction.

The latency plot reveals another weak point of the use of a handshaking protocol in combination with a single threaded forwarder. Since the communicating parties have to wait on their handshake, they at least have to wait for a normal MPI transfer of 7 integers for the information (depending on your C implementation about 28B) and the MPI transfer sending back the transfer ID of 1 integer (4B). Hereafter the normal communication is started. Since the reception from the sender and the forwarding to the receiver have to be done in serial, this adds up the transfer latency for that transfer size of both clusters. Latencies can even get worse if the forwarder handles another transfer in the mean time, which is to be expected when more than 2 parties engage in communication over the forwarder. For small transfers this can more than double the transfer latency found in the slower network, as can be seen in the graph.

For larger transfers, the latency of the MMPI implementation gets quite close to the theoretical best case for transfer in serial, which is the time taken to send the packet from the sender to the forwarder plus the time taken to send the data from the forwarder to the receiver. The overhead of the handshake will be relatively small compared to the transfer time of larger amounts of data, so is ignored in this best case.

Several interesting bumps are present in the latency graph. The first one is around transfers of 100B, which is a clear result of the underlying native MPI implementation that also expresses this bump. The other interesting bump can be found at transfers around a 256KB transfer size. It is mainly interesting because it only shows up in the MMPI implementation and it is also present in the form of a dip at the 256KB transfer size in both the bandwidth plots. One can only speculate why it is there, but the fact that the performance restores at higher transfer sizes and the size of the dip seem to rule out that it has anything to do with the L2 cache of the system which is 256KB, and might cause cache misses. There is not enough information to be sure, so this is for future investigation.

3.4 Barrier

The goal of `MPI_Barrier` is synchronization by waiting till all processes have reached the barrier before continuing. The MMPI implementation of the barrier is straightforward. Both groups of processes perform a barrier on their local communicator, in which the forwarder is not included. As soon all processes in a communicator get through the barrier, the lowest ranked process will send a request to the forwarder. In contrary to the point-to-point case, the transfer ID will only be returned to the requesting process when the matching request has

been received by the forwarder. So it will block the calling processes until both groups of processes have reached the first barrier. After the transfer ID is returned to both processes, the processes will again perform a barrier in their local groups. To make an approximation for the case in which the barrier latency is

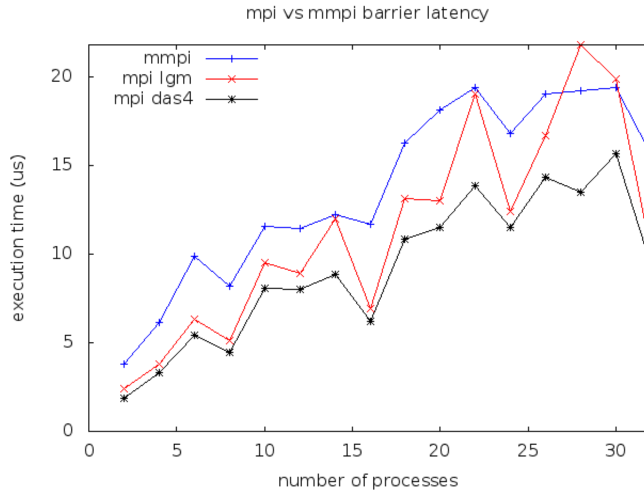

Figure 7: Latency of both the MMPI and the MPI barrier implementations.

much higher than the 4 byte message latency, let n be the amount of total processes in both process groups, let $barriers_{1/2n}$ be the normal MPI barrier time in the slow group for $n/2$ processes and let $barrierf_{1/2n}$ be the normal MPI barrier time in the fast group for $n/2$ processes. Since two barriers are performed with half of the total amount of processes in each local group, the expected latency of the MMPI implementation in the slow group, in this case the LGM, will be about $2barriers_{1/2n}$. For the fast group, in this case the DAS4, the barrier time of the MMPI barrier will be about $barriers_{1/2n} + barrierf_{1/2n}$.

To see if this is the case, Figure 7 shows the latency of the normal MPI and MMPI implementations of `MPI_Barrier`. Note that the latencies are averaged over both the LGM and DAS4 latency time by nature of the benchmark. When taking this into account, the expected latency coincides quite well with the approximation.

A problem of algorithms that use two barriers is that they do not synchronize processes in two communication groups tightly [8], which might not be what you want if the purpose of the algorithm is synchronization.

3.5 Broadcast

`MPI_Bcast` is used to send a message from one process, the root, to all the other processes in a communicator. In MMPI, the root sends a request to the forwarder

to make a broadcast. It uses the ID that it received during the handshake as a tag to send the buffer to the forwarder. After that, it starts a native `MPI_Bcast` with the forwarder as root. Note that this time only the root does the handshake, all the other processes of both clusters immediatly perform the native `MPI_Bcast`.

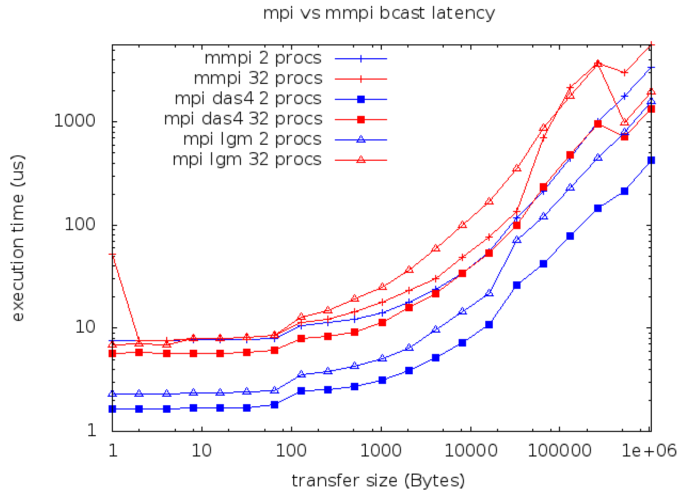

Figure 8: Latency of both the MMPI and the MPI broadcast implementations for different transfer sizes.

In Figure 8 it can be seen that the performance drop for low transfers, caused by the handshaking protocol, is present in MMPI's `MPI_Bcast` at least in case of 2 processes. However when comparing and extrapolating the results in Figures 8 and 9, for transfer sizes between 100B and 100KB among more than 4 processes, MMPI's `MPI_Bcast` latency is about as high as it is for the normal MPI implementation in the worst cluster up to the average time of the native MPI implementation in both clusters. It is hard to pinpoint an exact reason for this behaviour and explanations may heavily rely on the broadcast algorithm used in Open MPI.

One explanation for the fact that MMPI gets the average latency of MPI's Bcast algorithms in both clusters at medium sized messages could be that the lowest ranked process in Open MPI's broadcast algorithm will not send much, for example only two messages. After these messages are sent, the forwarder can start the broadcast to the other cluster. This would cause a large overlap in the execution of the broadcast calls in both clusters, keeping latency down. However, this theory can not explain why this low latency does not scale to transfer sizes higher than 100KB, so future research is needed.

Another set of performance degradations can be found in Figure 8 at sizes of transfer which correspond with the cache sizes of the system. One is quite small and is around the the L1 cache of 32KB in the 2 processors case. The

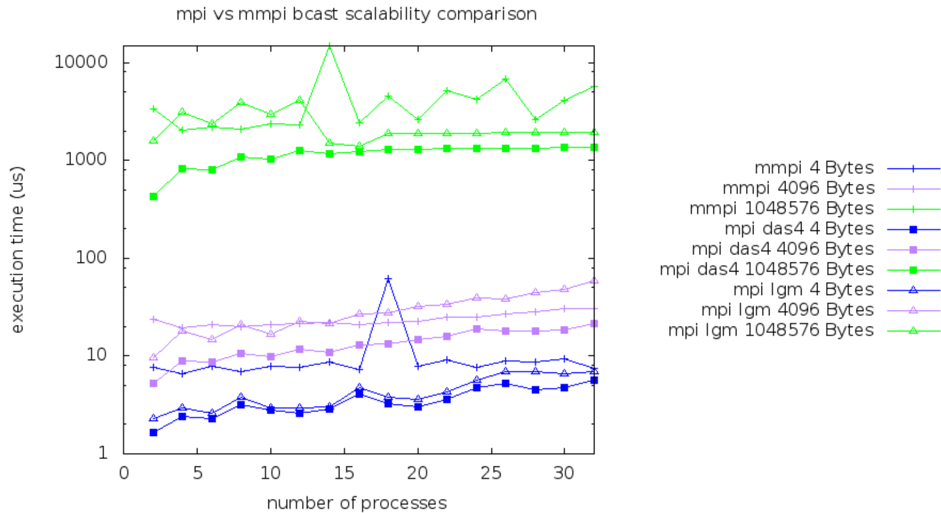

Figure 9: Latency of both the MMPI and the MPI broadcast implementations for different amounts of processes.

other one appears around transfers as big as the 256KB L2 cache. As discussed in section 3.3, it is impossible to prove with our results that these are indeed the result of cache misses, so it is for future research to confirm this.

3.6 All to All

The `MPI_Alltoall` and `MPI_Alltoallv` subroutines send messages from all processes to all other processes. The first one handles fixed sized message, the second one handles variable sized messages.

The MMPI implementation basically handles sends and receives from processes within the local group using the native MPI all to all subroutines. To reach the processes in the remote group, the MMPI versions of the non-blocking sends and blocking receives are used. Since each call generates overhead because of the repeated handshake protocol, a large overhead is expected for small transfer sizes. This can be clearly seen in Figures 11 and 13 for the 4B transfer size, but is more clear in Figures 10 and 12.

Another result can be found by comparing these figures. There is quite a performance difference between the MPI versions of
`MPI_Alltoall` and `MPI_Alltoallv`, especially for small messages with a relatively large number of nodes. Apparently this increased performance does not help the MMPI implementation, since the curves of `MPI_Alltoall` and `MPI_Alltoallv` are nearly the same. This again demonstrates the loss in performance for small messages due to the handshaking protocol.

Figures 11 and 13 show that, after performing as well as the MPI versions

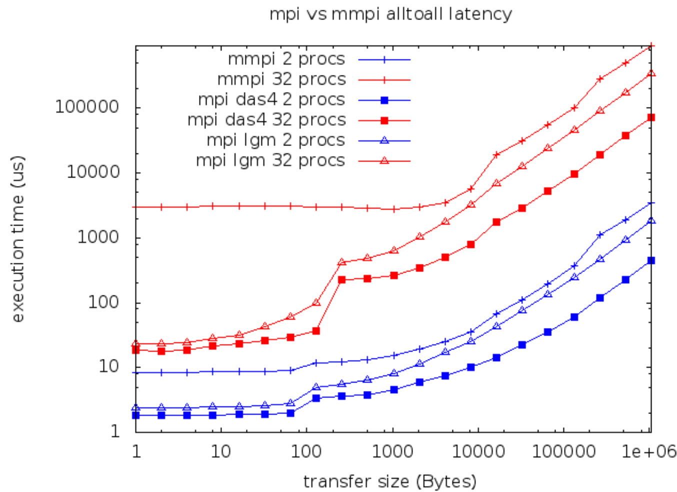

Figure 10: Latency of both the MMPI and the MPI alltoall implementations for different transfer sizes.

in the worst network at 4096B, another performance drop for the higher transfer size of 1MB appears. Since the relative overhead of the handshakes will be less, it does not account for this drop. However, making a quick calculation, for 32 processes, the fowarder needs to forward 32∗16MB = 512MB. Although this is less than the 550MB found in the bidirectional bandwidth plot of Figure 6, the plot has been made with only 2 processes communicating. In this case, 32 processes are using the forwarder. Although these figures do not provide direct proof for it, it is well known that when using other than ping-pong communications, the performance of the InfiniBand will decrease significantly [14], which without a doubt will influence the bandwidth available to the forwarder, probably leading to a lower forwarding bandwidth than 550MB/s. This would again point out that the half-duplex property of the forwarder is a bottleneck.

3.7 Allreduce and reduce

MPI's allreduce and reduce operations take data from all the processors and perform a calculation on it, like a summation or taking the maximum or the minimum. If more than one element is provided, the calculation will be done in a per-element fashion. For instance, if 4096 bytes are given to the function with the summation operator, the first byte handed over by all processes will be summed and stored in the first byte of the result. The same is done for the second byte, till all bytes have been processed and a result of 4096 bytes has been generated. In the case of reduce, only one of the processes, named root, will receive the result. In the case of allreduce, everybody receives the result of the calculation.

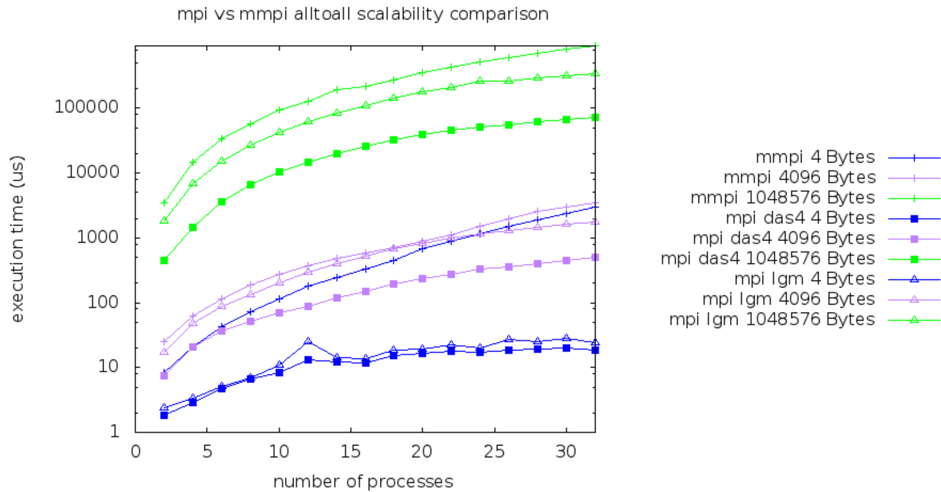

Figure 11: Latency of both the MMPI and the MPI alltoall implementations for different amounts of processes.

The MMPI implementations are quite similar. Both start performing a native `MPI_Reduce` operation. In the reduce version, in one group the root, which has received the result of the reduce operation, will handshake with the forwarder. In the other group the lowest rank, provided with the reduce result of that group, will handshake with the forwarder. In allreduce, the lowest ranks of both groups will do the handshaking and the reception of the result. After this, the parties who did the handshaking will send the result to the forwarder, which calculates the final result by combining the information. In reduce, the forwarder will send the final result to the root. In allreduce, the final result is broadcasted by the forwarder over all participating processes using a native `MPI_Bcast` function.

The evaluation of reduce can be found in Figure 14. The graph is not very clear because it shows the averaged results of the execution time of all processes. In MMPI's reduce implementation, the root process has to wait a lot longer than the other processes, but this effect can be averaged out if the other processes have lower latencies than in the native MPI version. In Figure 15, this is probably the reason why the different curves lie so close at higher numbers of nodes.

It is better to look at results that are less averaged out. Looking at the results for two processes in Figure 14 and comparing it with the latency plot of Figure 6 shows us, that sending the information to the forwarder and receiving the information from the forwarder accounts for nearly all of the latency. This is because the reception of the intermediate results from both groups is happening using a blocking receive call, which makes the whole process of receiving from the groups and sending it back to the root serial. Changing these receives into non-blocking ones will probably result in a significant reduction in latency for this operation.

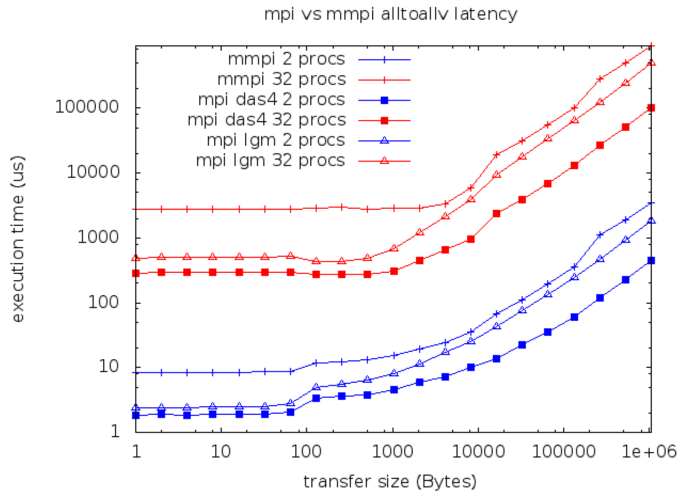

Figure 12: Latency of both the MMPI and the MPI alltoallv implementations for different transfer sizes.

The results for the allreduce algorithm, found in Figures 16 and 17 are less averaged out since all processes are blocking till they get the result from the broadcast. For the medium sized messages, the performance seems to scale well regarding the amount of processes that are active in the operation. For the 4B transfer size and the 1MB transfer size however, there seems to be some overhead. In the 4B case, nearly all the overhead seems to result from two times the broadcasting and two times the local allreduce. Looking at the 1MB case, the MMPI curve can not fully be explained by adding up the send latencies and the broadcast latencies. It also does not explain the undulating character of Figure 17. However, finding another explanation is hard.

In Figure 14 and Figure 16, again a peak can be found at 256KB message sizes and a smaller one can be found around 32KB. This time they actually might be due to an increase in cache misses, since the forwarder needs to see every byte and perform the reduction operation. An inefficient scheme in the forwarder is used in which a special result buffer is separately allocated, instead of reusing one of the two receive buffers. Since at sizes bigger than 8kB the three buffers do not fit in one L2 cache anymore, this will lead to an increase in cache misses. However, in the allreduce graph, the latency is about 20 times higher than the latency expected if you would interpolate the latency. If this is due cache misses, an increase in latency of similar order would be expected at this point in Figure 14. So the effect might be too big to be caused by L2 cache misses alone. It would be interesting to see whether the increase in latency is truly due to cache misses during future research.

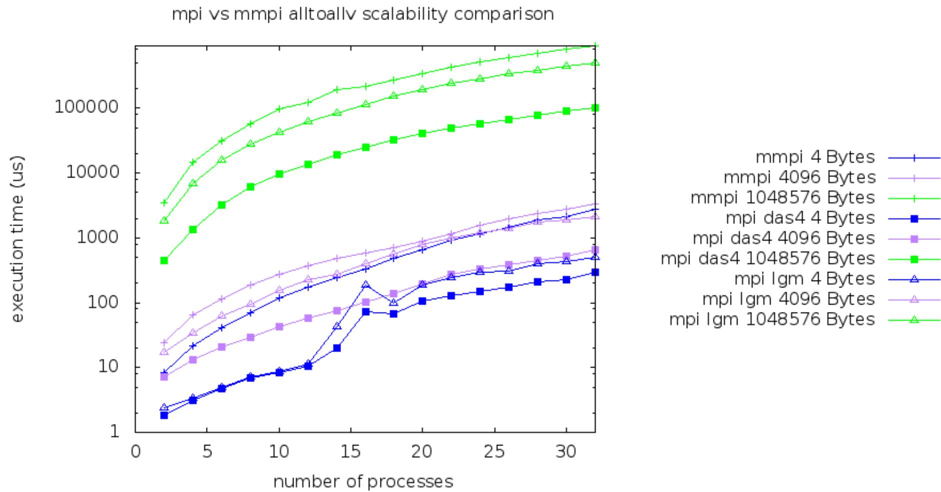

Figure 13: Latency of both the MMPI and the MPI alltoallv implementations for different amounts of processes.

4 NAS Parallel Benchmarks

To get a more realistic indication of how MMPI would perform, when used in a real application, it was evaluated using a subset of the NAS Parallel Benchmarks (NPB) version 3.2. This benchmark suite consists of several kernels that are frequently used in computation fluid dynamics and some simulated application benchmarks which try to mimic computation in actual computational fluid dynamic applications [2]. From this benchmark, 4 kernels were chosen: EP, MG, CG and IS, which will be respectively presented with their results in Sections 4.1, 4.2, 4.3 and 4.4. Apart from the kernels, one simulated application was chosen from these benchmarks, LU, which is presented along with the results in Section 4.5.

The NPB provides several sizes of the benchmark. For this cluster, size class C was chosen because it gives significant calculation times and scales well when adding more nodes.

It must be noted that the benchmark results for the LGM end at 32 nodes since there were not enough free nodes available at the time these benchmarks were ready. All presented results are averaged over 15 runs.

4.1 EP

EP is an 'Embarassingly Parallel' application that performs minimal communication and therefore is a good indication of the throughput of the system.

As described in Section 2.3 the processors of both DAS4 and LGM should be the same, like much else of the system, so it is quite a surprise that, when looking at Figure 18, the LGM is about 20% faster than DAS4 at low numbers

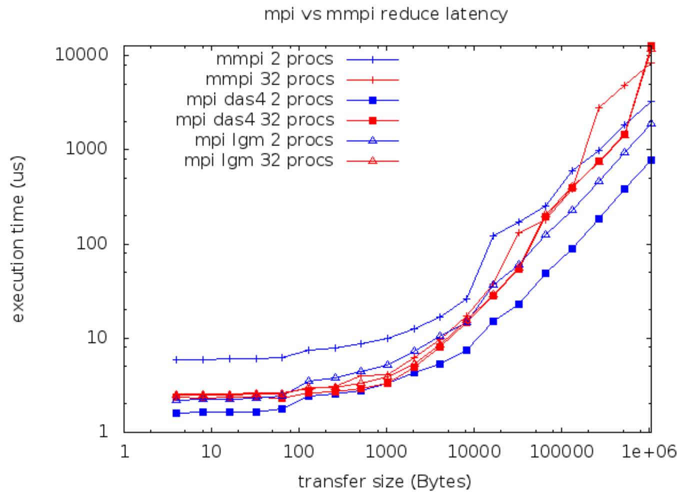

Figure 14: Latency of both the MMPI and the MPI reduce implementations for different transfer sizes.

of processes. Less surprising is the fact that the MMPI implementation has a speedup compareable to the slowest of the two clusters, namely the DAS4. After 48 nodes, the MMPI version starts to perform worse. This occurs at the moment an extra compute node in both clusters is added to the calculation. Since the benchmark tests throughput of the system, overhead due to MMPI is unlikely. Probably someone had programs running on this node which interefered with the measurements.

4.2 MG

MG is a MultiGrid kernel. It is a computation bound kernel (about 80% of the running time) [21], but it does perform some communication. The bulk of its traffic consists of blocking sends and non-blocking receives, but `MPI_Allreduce` is used frequently enough (about once on 100 sends) to be mentioned [20]. The point-to-point communications are done in a ring topology [17]. With the process distribution used in these experiments, this will keep a lot of the communication restricted to the local clusters. The results presented in Figure 19 show us that for small communication loads, MG can scale quite well with the number of nodes and only performs slightly worse than MG executing on the DAS4. Reason for this is that the badly scaling allreduce operation is not used much, and only for transfer sizes smaller than 4 integers (16B on some implementations of C).

Pinpointing the problem therefore is quite hard, but since the slowdown compared to the case where MPI is running on DAS4 is small, the bottleneck probably lies within the bandwidth of the forwarder. Because of the ring topology, 2 pairs

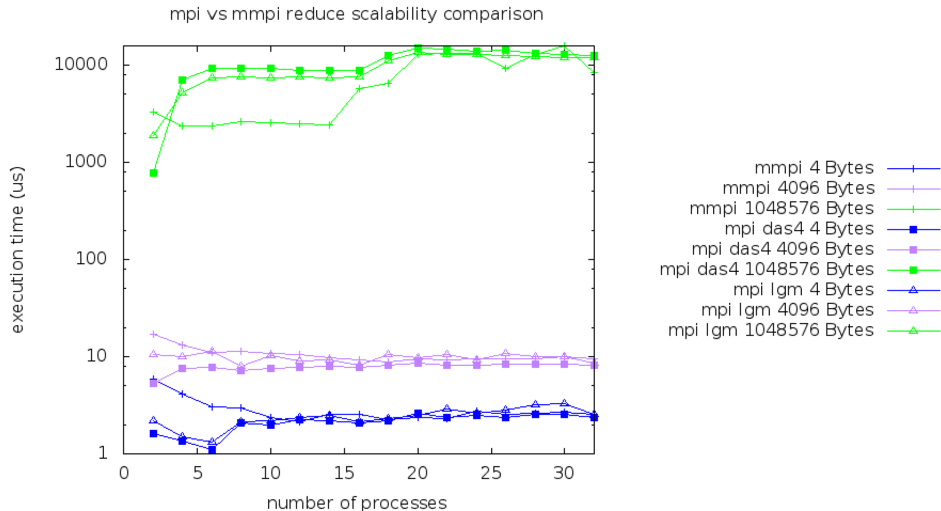

Figure 15: Latency of both the MMPI and the MPI reduce implementations for different amounts of processes.

of processes will cause the bulk of the communication over the forwarder. Like mentioned in Section 3.6, the bandwidth of the traffic that the forwarder can forward will probably drop and the average latency of transfers will increase as the number of transfers from different processes increases. This explanation would be in line with other studies which say that bandwidth is more important for the MG benchmark than latency [19].

4.3 CG

CG is a Conjugate Gradient method which performs sparse matrix-vector operations. Like MG, most of CG's operations are point-to-point communication in which it moves even more data than MG [6]. CG communicates with a chain like structure, meaning process 0 communicates with process 1, process 1 communicates with process 0 and 2, etc [17]. This means that most traffic over the forwarder will be generated by the same two processes.

Our results confirm this extra pressure on the interconnects. When looking at Figure 20, up to 16 nodes, the LGM is superior to both MMPI and DAS4. This is clearly the case because LGM has the superior throughput as can be seen from the EP results in Figure 18. When more nodes are added to the calculation, and more traffic is produced, the scaling of the application seems to become very bandwidth dependent. If we compare the bidirectional bandwidth for the different clusters found in Figure 6 with the results found in Figure 20, we can see that the version allowing the fastest process communication, performs best.

Since the performance of MPI on LGM is not known beyond 32 nodes, it

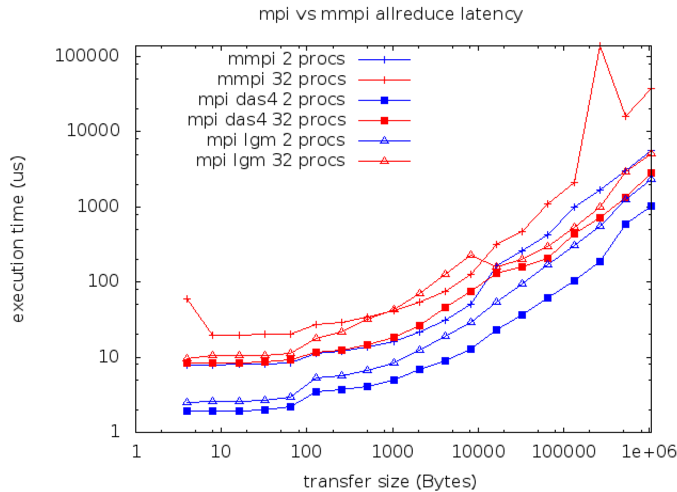

Figure 16: Latency of both the MMPI and the MPI allreduce implementations for different transfer sizes.

would be interesting to see if adding an extra forwarding process to provide extra bandwidth would make the performance of MMPI similar to that of MPI on LGM.

4.4 IS

IS is an Integer Sorting application which is quite different from the other benchmarks. Not only because it mainly does integer arithmetic instead of floating point arithmetic, but it is different in a communication perspective. IS's computation mainly consists from collective operations (`MPI_Allreduce`, `MIP_Alltoall(v)` and `MPI_Reduce`), and it has quite a lot of big transfers, with big being bigger than 1MB up to a about 30MB [20]. Not suprisingly, IS is reported to be communication bound [21]. With that knowledge, Figure 21 contains no surprises. An interesting fact is that MMPI has about the same factor of slowdown in respect to MPI on LGM for small numbers of nodes when comparing the bi-directional bandwidth plot in Figure 6. However, with increasing numbers, the performance of MMPI scales a lot worse than MPI on the LGM, which has the worst network.

This could be the result of the bad scaling of MMPI's allreduce. Another possible reason is, that the bandwidth of the forwarder collapses a lot faster for a large number of nodes trying to communicate than standard InfiniBand like described in Section 3.6. This is important since alltoall is an important part of IS.

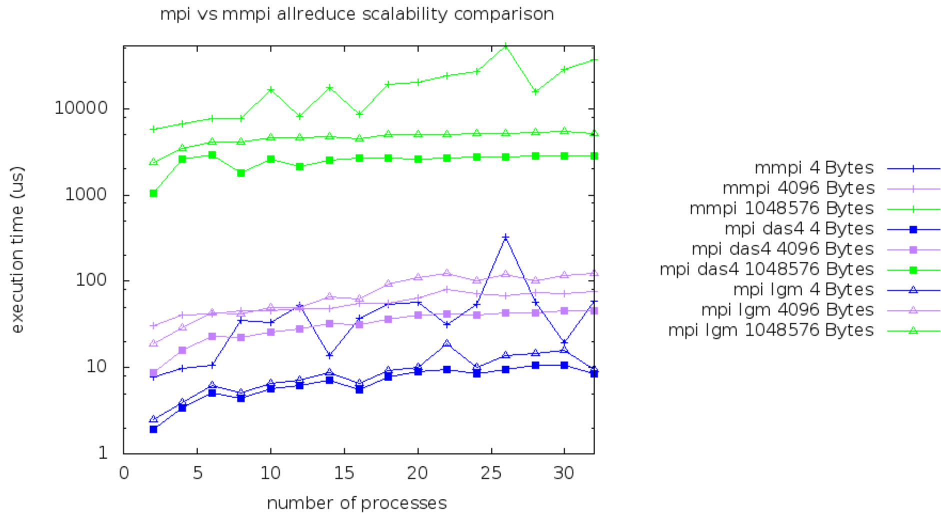

Figure 17: Latency of both the MMPI and the MPI allreduce implementations for different amounts of processes.

4.5 LU

The LU benchmark is a simulated application and therefore the closest thing to a real computational fluid dynamic application we get in this study. It generates several GB's of traffic through point-to-point communications [6]. Like MG it communicates in a ring-topology, restricting much of the communication to the clusters in which the nodes are present. The difference with MG is that the message rate of LU is lower [17]. Looking at Figure 22, the break-even point of communication and computation seems to be at 32 nodes. Here MPI and the MMPI version on both clusters have about the same runtime. It is hard to tell if the performance of MMPI on 64 nodes is bad due to the network performance of LGM, or due to overhead of the MMPI implementations.

5 Related Work

Although the architecture of the 'Leiden Zipper' seems to be unusual, the solution of using repeaters to forward traffic between clusters is not. Both BC-MPI [5] and PAXC-MPI [8] use separate forwarding processes, in the latter called daemons, for inter-cluster traffic. The forwarder that is presented in Section 3 is similar to the PAXC-MPI implementation in many other ways, like for instance in the way barriers are implemented, how virtual/global process numbering is done and the fact that it is an implementation on top of a native MPI implementation.

Many other MPI implementations support inter-cluster communication, like MPICH-G2 [10] and MagPIe [12]. However, to the author his knowledge, this is

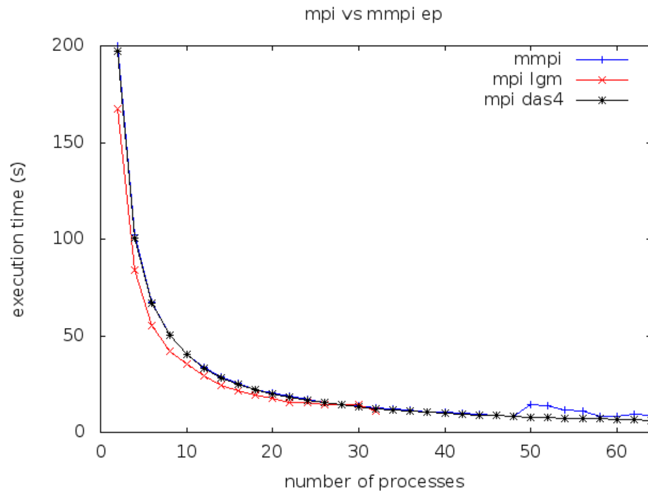

Figure 18: Comparison of Open MPI on LGM and DAS4 with MMPI for the EP benchmark.

the first study describing the forwarding of data between two InfiniBand networks by using software.

6 Discussion

The results presented in Section 3 have exposed two elements of the design which significantly decrease the performance of the forwarder. The first one is the handshake protocol, which in several cases degraded the performance of MPI operations at low transfer sizes. The second one being the fact that the forwarder is only half duplex, decreasing the possible bandwidth.

Another, smaller, decrease in performance is caused by the blocking receives used, for instance, in Allreduce. Making these non-blocking would increase performance since they can be executed in parallel.

Furthermore the LU and MG benchmarks show us that with equal distribution of nodes in both clusters, the combined cluster only scales as well as the worst of the two. This limits the possibilities of deploying zipped nodes. Too slow interconnects will create big performance penalties.

Applications with relatively few communication over the 'Leiden Zipper' perform quite well, as can be seen for LU, MG and EP. IS is a nice example of a communication pattern (all nodes communicate with each other) of applications that will not work well. However, from this study, it is hard to see how MMPI and the zipper architecture perform in comparison to other grid enabled MPI versions.

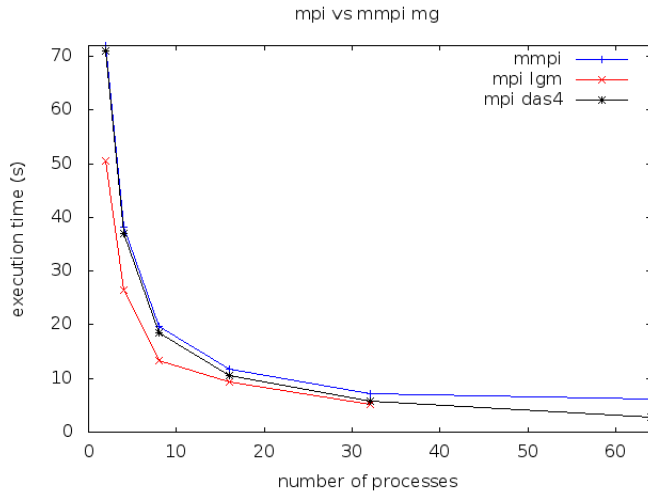

Figure 19: Comparison of Open MPI on LGM and DAS4 with MMPI for the CG benchmark.

7 Conclusion

This thesis describes an alternative way of connecting clusters into a grid by connecting both clusters to a set of zipper nodes. It is shown that applications that have a communication graph with either a cut-set or a vertex-cut set can in theory be run without extra overhead of routers or forwarders on the 'Leiden Zipper'.

For applications which do not exhibit this ideal property, information must be forwarded. For this purpose, a layer on top of Open MPI, which routes inter-cluster traffic over a forwarding process, MMPI, is described and evaluated. Lessons learned from the evaluation of the design include the overhead caused by the handshake protocol for small transfers and the half-duplex property of the forwarder.

Results from the NAS Parallel Benchmark show that applications with low communication frequencies and ring communication topologies like LU, scale well on the Leiden Zipper. Communication intensive applications like IS do not.

Furthermore it is shown on the Leiden Zipper architecture that running applications from the NAS Parallel Benchmark with an even distribution of processes over both clusters, will not scale better than the worst cluster.

8 Future Work

Based on the results presented in this thesis, several things are interesting to investigate of which the most obvious one is improving MMPI. Just plain practical

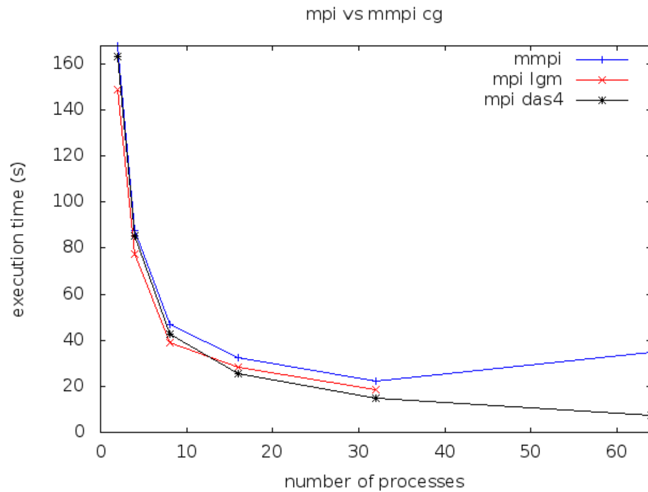

Figure 20: Comparison of Open MPI on LGM and DAS4 with MMPI for the CG benchmark.

improvements would include implementing the existing functions in conformance with the MPI standard and implementing more functions from the standard. This would make applications source-code compatible, which would remove the need for rewriting.

Also some improvements can be made on the performance of MMPI. An alternative has to be found for the handshaking algorithm, as it gives a lot of overhead. As for the relatively low bandwidth of MMPI, different alternative solutions could be evaluated like multiple forwarders and maybe multithreaded forwarders. Having multiple forwarders might also solve part of the increase of latency due to the forwarding being done serial, like described in Section 3.3, since message striping schemes are possible which increase the available bandwidth for one message.

It would also be interesting to investigate the theory of loss of performance due to the influence of cache misses, as discussed in Section 3.3. Some hardware counter statistics could either confirm or falsify this theory. If the theory is true, it might be possible to increase performance by implementing cache-friendly algorithms in the forwarder.

Other things to investigate are applications that have communication graphs with either vertex-cut sets or cut sets, as is discussed in Section 2.2, or communication graphs that are very similar. This thesis only presents a theoretical result about these ideal types of applications. The question whether it is possible to rewrite such applications so that there is no or very little overhead, perhaps automatically, remains open.

If rewriting is not possible, it might be possible to map applications to the

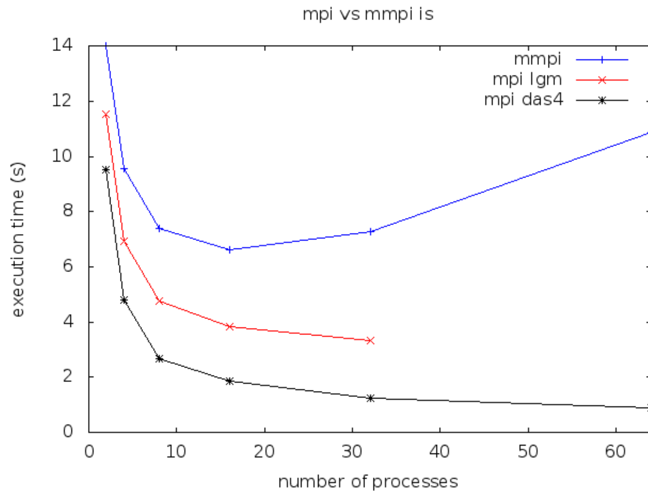

Figure 21: Comparison of Open MPI on LGM and DAS4 with MMPI for the IS benchmark.

zipper architecture automatically for maximal performance. This could allow applications mapped over the zipper architecture to outperform the worst of the zipped clusters, which was not possible by the process mapping presented in Section 3.1, or at least not for the tested benchmarks. Another challenge considering process mapping might lie in the hardware differences among clusters. If some processes need a GPU, which might not be available among all nodes of both clusters, is it possible to automatically map these processes to nodes which contain a GPU?

Finally, this thesis does not compare the MMPI solution with other existing solutions. A comparison with for example RDMA over Ethernet and existing grid enabled MPI applications could shine a light on other techniques that allow optimal use of the 'Leiden Zipper' architecture in the field of grid computing.

9 Acknowledgements

A word of thanks goes out to both my supervisors, Prof. Dr. Harry Wijshoff and Drs. Kristian Rietveld, for the input and support during the project. I also would like to thank Vianney Govers for his time invested answering the many questions I had about the hardware of the system and for providing the Dutch version of Figure 1.

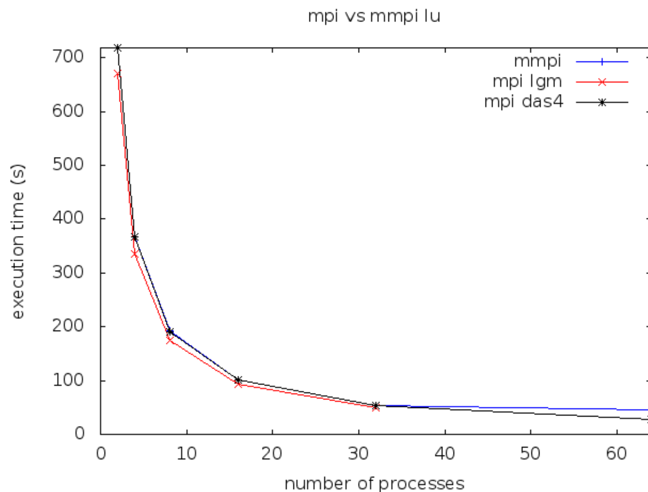

Figure 22: Comparison of Open MPI on LGM and DAS4 with MMPI for the LU benchmark.

References

[1] InfiniBand Trade Association et al. Infiniband architecture specification, vols 1 & 2, release 1.2, october 2004, 2004.

[2] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.

[3] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[4] Madhu Chetty and Rajkumar Buyya. Weaving computational grids: How analogous are they with electrical grids? *Computing in Science & Engineering*, 4(4):61–71, 2002.

[5] Paweł Czarnul. Bc-mpi: running an mpi application on multiple clusters with beesycluster connectivity. In *Parallel Processing and Applied Mathematics*, pages 271–280. Springer, 2008.

[6] Ahmad Faraj and Xin Yuan. Communication characteristics in the nas parallel benchmarks. In *IASTED PDCS*, pages 724–729, 2002.

[7] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[8] Edgar Gabriel, Michael Resch, and Roland Rhle. Implementing mpi with optimized algorithms for metacomputing, 1999.

[9] Steve Huss-Lederman, Bill Gropp, Anthony Skjellum, Andrew Lumsdaine, Bill Saphir, Jeff Squyres, et al. Mpi-2: Extensions to the message-passing interface. In *University of Tennessee, available online at http://www. mpi-forum. org/docs/docs. html*, 1997.

[10] Nicholas T Karonis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[11] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, 3(4):267–286, 1979.

[12] Thilo Kielmann, Rutger FH Hofman, Henri E Bal, Aske Plaat, and Raoul AF Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *ACM Sigplan Notices*, volume 34, pages 131–140. ACM, 1999.

[13] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

[14] Subhash Saini, Andrey Naraikin, Rupak Biswas, David Barkai, and Timothy Sandstrom. Early performance evaluation of a nehalem cluster using scientific and engineering applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 24. ACM, 2009.

[15] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[16] Heinz Stockinger. Defining the grid: a snapshot on the current view. *The Journal of Supercomputing*, 42(1):3–17, 2007.

[17] Jaspal Subhlok, Shreenivasa Venkataramaiah, and Amitoj Singh. Characterizing nas benchmark performance on shared heterogeneous networks. In *ipdps*, volume 2, page 91, 2002.

[18] Hari Subramoni, Ping Lai, Miao Luo, and Dhabaleswar K Panda. Rdma over etherneta preliminary study. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–9. IEEE, 2009.

[19] Yuzhong Sun, Jianyong Wang, and Zhiwei Xu. Architectural implications of the nas mg and ft parallel benchmarks. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 235–240. IEEE, 1997.

[20] Theodore B Tabe and Quentin F Stout. The use of the mpi communication library in the nas parallel benchmarks. *Ann Arbor*, 1001:48109, 1999.

[21] Frederick C Wong, Richard P Martin, Remzi H Arpaci-Dusseau, and David E Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *Supercomputing, ACM/IEEE 1999 Conference*, pages 41–41. IEEE, 1999.