UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

High performance N-body simulation on computational grids

Groen, D.J.

Publication date
2010

Link to publication

Citation for published version (APA):
Groen, D. J. (2010). *High performance N-body simulation on computational grids*.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:23 Sep 2021

6 A Light-Weight Communication Library for Distributed Computing

Based on:
D. Groen, S. Rieder, P. Grosso, C. de Laat and S. Portegies Zwart,
A Light-Weight Communication Library for Distributed Computing.
Computational Science and Discovery, vol. 3, art. no. 015002, August 2010.

In this chapter we present the MPWide communication library, which we developed to facilitate high performance computing across multiple supercomputers.

6.1 Introduction

A parallel application can run concurrently on multiple supercomputers provided one is able to coordinate the tasks between them and limit the performance overhead of the wide area communications. The advantage of using a distributed infrastructure lies in the enormous amounts of storage, RAM and computing performance it makes available. Distributed computing therefore allows us to solve large scale scientific problems [60]. Starting from the coupling of Intel Paragons over an ATM network [112] in the early 1990s, distributed parallel applications have become very popular.

An efficient method to program a parallel application is the Message Passing Interface (MPI [118]), a language-independent communication protocol that coordinates the computing tasks in parallel programs. MPI is often used for intra-site parallelization, but it can also be used for message passing in a distributed infrastructure. In this case, processes exchange data with their local peers, as well as processes at other sites. Prior efforts in the use of MPI on distributed infrastructures are abundant [57, 89, 12] and several implementations have emerged which support execution across sites [37, 70]. With respect to N-body simulations Gualandris et al. [46] have demonstrated that it is possible to use grid-enabled clusters of PCs connected via regular internet, grid middleware and MPICH-G2 [70]. However, the vast majority of MPI implementations require

all participating nodes to have public IP addresses, which is generally undesirable for supercomputer environments for security reasons. Furthermore these implementations do not have a built-in optimization to fully exploit dedicated network circuits, a central component in multi-supercomputer infrastructures.

The lack of flexibility in deployment and link-specific optimizations of grid-oriented MPI implementations in distributed supercomputer environments led us to develop MPWide, a light-weight socket library specially aimed for wide-area message passing between supercomputers. In this paper we present several performance results and apply MPWide to parallelize a large-scale cosmological N-body simulation across two supercomputers.

6.2 Related work

Several grid message-passing libraries and frameworks have been developed with the intent to make distributed computing possible between sites that have restrictive firewall policies. PACX-MPI [38] is specifically geared for parallelization across sites and does not require compute nodes to have a public IP address. Instead, it forwards inter-site communications through two forwarding demon processes on each site. Such a setup works reasonably well for applications that have been parallelized over multiple supercomputers using regular internet [121], but the restriction to two communication processes is less optimal when using multiple sites in a dedicated network environment. The Interoperable MPI (IMPI) [41] standard has also been designed to specifically facilitate execution across sites, but at the time of writing very few of the vendor-tuned implementations on supercomputers support IMPI. Also, IMPI requires the installation of a centralized and globally accessible server and does not support path-specific optimizations.

NetIbis [11] and PadicoTM [26] are two communication frameworks which are able to establish connections using bootstrap links, thus not requiring public IP addresses. However, PadicoTM also requires the use of a centralized rendez-vous node for bootstrapping, and thereby some means of centralized connectivity. Both Ibis [127] and NetIbis are sufficiently flexible to use in a restricted supercomputer environment, but introduce a communication overhead compared to regular socket communications. These libraries are therefore less suitable for high-performance message passing over dedicated inter-supercomputer networks.

6.3 Architecture of MPWide

6.3.1 Design

MPWide is a light-weight communication library which connects multiple applications on different supercomputers, each of them running with the locally recommended MPI implementation. It can be installed by a local user without administrative privileges, has a very limited set of software requirements, and the application programmer interface is

similar to that of MPI. The applications are deployed separately for each supercomputer, and use MPWide to connect with each other upon startup. We are considering to add support for automated deployment, but to accomplish this we require a method to initiate applications on remote sites. The development of such a mechanism is not straightforward, because the access and security policies tend to be different for each supercomputer.

MPWide has been designed to facilitate message passing between supercomputers and construct/modify custom communication topologies. The MPWide library is linked to the application at compile time and requires only the presence of UNIX sockets and a C++ compiler. MPWide provides an abstraction layer on top of regular sockets with methods to construct a communication topology, to adjust the parameters of individual communication paths and to perform message passing and forwarding across the topology. MPWide does not link against local MPI implementations, but can be used to combine multiple programs parallelized with MPI. Maintaining separate implementations for intra- and inter-site message passing makes it easier to specifically optimize and debug long-distance communication paths while relying on well-tested and vendor-tuned software for optimal intra-site communication performance.

6.3.1.1 Data transport in the wide area network

Dedicated network circuits are excellently suited for facilitating data transport between supercomputers. The highly deterministic bandwidths of optical circuits (or *lightpaths*) reduce the communication time while properly tuned transport layer protocols increase the application throughput in the absence of competing traffic.

During the development of MPWide, we have examined the communication performance of several protocols by transferring data between two nodes in the Netherlands using a 10 Gbps optical network that was looped via Chicago, USA. We ran tests using both the TCP and the UDP network protocols. Plain UDP does not ensure the integrity of data packets however, and is therefore unsuitable for message passing. As an alternative, we instead tested the performance of two modified UDP implementations which feature mechanisms to ensure data integrity. These are Reliable Blast UDP (RBUDP [52], which is part of the Quanta toolkit [51]) and the UDP-based Data Transfer protocol (UDT [44]). The tests using TCP were run with both a single communication stream and with multiple streams in parallel.

We achieved a network throughput of less than 1 Gbps using RBUDP or UDT, and a throughput of up to 6 Gbps using parallel TCP streams. A full technical report on these preliminary performance tests can be found in [21]. Based on these results we decided to rely on multiple streams with a TCP-based protocol. This is a well-known and proven techniques to improve network performance in the WAN [114].

6.3.1.2 Functionality and programming interface

In MPWide, the communication takes place through *channels*. Each channel makes use of a single socket and provides a bidirectional connection between two ports on two hosts.

On network paths where the use of parallel TCP streams provides a performance benefit, it is possible to use multiple channels concurrently on the same path. The message passing and forwarding functions in MPWide are designed to operate concurrently on multiple channels when needed.

Channels are locally defined at initialization and may be closed, modified and reopened at any time during execution. This allows us to alter the communication topology at run-time, for example to restart or migrate part of the MPWide-enabled application.

Once one or more communication channels have been established, the user can transfer data using the communication calls in the MPWide API. Table 6.1 provides an overview of the functionality provided by MPWide.

command name	functionality
`MPW_Barrier()`	Synchronize between two ends of the network.
`MPW_Cycle()`	Send buffer over one set of channels, receive from other.
`MPW_DSendRecv()`	Send/receive buffers of unknown size using caching.
`MPW_Init()`	Set up channels and initialize MPWide.
`MPW_Finalize()`	Close channels and delete MPWide buffers.
`MPW_Recv()`	Receive a single buffer (merging the incoming data).
`MPW_Relay()`	Forward all traffic between two channels.
`MPW_Send()`	Send a single buffer (splitted evenly over the channels).
`MPW_SendRecv()`	Send/receive a single buffer.

Table 6.1: List of MPWide function calls. In addition to this list, each function has a variant call with a prefix 'P' which operates on one send and/or recv buffer per channel.

Since message passing can be performed over multiple channels in parallel, it is possible to communicate with multiple hosts simultaneously. For example, the user can scatter data across multiple processes with a single `MPW_Send()` call or gather data from multiple hosts with a single `MPW_Recv()`. Each function has a variant call with a prefix 'P' (e.g., `MPW_PSend()`) which takes an array of buffer pointers instead of one buffer pointer. These functions use one pointer for each channel, and the size of each separate buffer can be explicitly specified. Consequently, `MPW_PSend()` or `MPW_PRecv()` functions can be used to respectively scatter and gather data which is not equally distributed across the hosts.

Both `MPW_Cycle()` and `MPW_DSendRecv()` also support the receiving of data buffers which are of unknown size (but not larger than a given size limit provided by the user). This feature may provide some performance improvement in long distance environments at the expense of possible excessive memory consumption, as separate calls to exchange data size information are no longer required

An MPWide code example is shown in Fig.6.1. There we initialize MPWide with one single-stream path to a local network address, the `LANChannels`, as well as a double stream path to a different site, the `WANChannels` (lines 1-12). The program then initializes two buffers (line 14-15); it reads 100 bytes of data from the local connection

```
1  int NumChannels = 3; // Total number of channels.
2  int NumLAN      = 1; // Number of LAN channels.
3  int NumWAN      = 2; // Number of WAN channels.
4  int MsgSize     = 100;
5
6  int Hosts = {"10.0.0.100","123.45.67.89", "123.45.67.89"};
7  int Ports = {6000, 6001, 6002};
8
9  MPW_Init(Hosts, Ports, NumChannels);
10
11 int LANChannels[NumLAN] = {1};
12 int WANChannels[NumWAN] = {2,3};
13
14 char* SendBuf = new char[MsgSize];
15 char* RecvBuf = new char[MsgSize];
16
17 // Recv from LAN.
18 MPW_Recv    (SendBuf, MsgSize, LANChannels, 1);
19 // WAN exchange.
20 MPW_SendRecv(SendBuf, MsgSize, RecvBuf, MsgSize, WANChannels, 2);
21 // Send to LAN.
22 MPW_Send    (RecvBuf, MsgSize, LANChannels, 1);
23
24 /* ( ... Process data and delete SendBuf and RecvBuf. ... ) */
25
26 MPW_Finalize();
```

Figure 6.1: Example code of the MPWide functionality

with `MPW_Recv()` (line 18); it exchanges this data with the remote WAN communication node using `MPW_SendRecv()` (line 20). At this point, the program has received the data from the remote WAN node, and forwards this data to the local connection (line 22).

6.3.2 Forwarder

When running an application across multiple sites, the processes on one site are not always directly able to communicate with the other site. In many cases this problem can be resolved by forwarding the messages to intra-cluster communication nodes, which do have access to all other sites through the wide area (dedicated) network. However, when the application uses a topology containing multiple dedicated networks, it will be necessary to forward messages from one network to another. The `MPW_Relay()` function provides such message forwarding for MPWide channels, and has been incorporated into the MPWide Forwarder. The Forwarder provides message forwarding for MPWide

in user space, connecting an MPWide channel from one network to that of another. It can therefore serve as relay process between nodes that are otherwise unable to contact each other or be put on intermediate nodes on very long network lines to mitigate the performance impact of packet loss. This latter method has been implemented and applied previously in the Phoebus project [7].

6.3.3 Implementation

We implemented MPWide using C++ in combination with GNU C sockets and POSIX threads [94]. MPWide creates and destroys threads on the fly whenever a communication call is made. With modern kernels, the overhead of creating and destroying threads is very small, and using MPWide we were able to reach nearly 10 Gigabit per second (Gbps) with message passing tests over local networks. For longer network paths, the high latency results in an even smaller relative overhead for thread creation/destruction. We have considered creating threads only at startup and managing them at runtime, but these modifications would increase the complexity of the code and only offer a limited performance benefit, as threading overhead plays a marginal role in wide area communication performance.

Aside from the ability to hardwire each communication, the library also supports a number of customizable parameters:

- number of concurrent streams for each communication call
- data feeding pace of sending and receiving.
- TCP window size for each individual socket

The maximum number of streams and the TCP window size may be restricted by local system policies. However, we were able to use up to 128 streams on most systems without requiring administrative rights. The code has been packaged and is publicly available at
`http://castle.strw.leidenuniv.nl/software/mpwide.html`.

6.4 Benchmarking MPWide

We have run a series of tests to measure the performance of MPWide between two local supercomputer nodes, as well as two nodes connected by a 10 Gbps international network connection. We have chosen not to compare the performance with that of other message-passing libraries. These libraries often require system level optimizations by administrators, while MPWide can largely be optimized in user-space. A direct comparison will therefore be biased, depending on the amount of optimization done by system administrators.

For the local tests, we use two nodes of the Huygens supercomputer in Amsterdam, the Netherlands [64], where the nodes are connected by 8 parallel Infiniband links, each of which supports a maximum bandwidth of 20 Gbps. Our local tests use one

out of these 8 Infiniband links. Each run consists of 100 two-way message exchanges, where we record the average throughput and the standard error. First we performed 8 different tests using messages of 8 MB and respectively 1, 2, 4, 8, 16, 32, 64 and 124 TCP streams in parallel. Due to system limitations, we were unable to perform tests using more than 124 streams on this particular site. We then repeated the same series of runs with message sizes of 64 and 512 MB.

The national tests were carried out between two sites of the Distributed ASCI Supercomputer 3 (DAS-3 [1]), one at the University of Amsterdam and one at the Delft University of Technology. Both sites are connected to regular internet with a 10 Gbps interface from the head node, and with a 1 Gbps interface from each compute node. A detailed specification can be found in columns 4 and 5 of Table 6.2. We performed the tests using the system default TCP window sizes (16 kB send and 85 kB recv).

For the international tests, we performed the same series of message exchanges, but now using one Huygens node and one node of the Louhi supercomputer in Helsinki, Finland [77], which are both connected to the DEISA shared network with a 10 Gbps interface. The round trip time of this network between Huygens and Louhi is 37.6 ms and we applied a TCP window size of 16 MB. The specifications of both supercomputers can be found in columns 2 and 3 of Table 6.2.

	Huygens	Louhi	DAS-3 Ams	DAS-3 Delft
CPU vendor	IBM	Cray	AMD	AMD
Architecture	Power6	XT4	Opteron	Opteron
Number of nodes	104	1012	41	68
Cores per node	32	4	4	2
CPU frequency [GHz]	4.7	2.3	2.2	2.4
Memory per core [GB]	4/8	1/2	1	2

Table 6.2: Specifications of the Huygens and Louhi supercomputers, as well as the two sites of the DAS-3 Dutch Grid.

6.4.1 Results

6.4.1.1 Local tests

The results of the local performance tests, performed in March 2009, are found in Fig. 6.2. The local network line has a very low latency (< 0.1 ms) and is therefore quickly saturated when using multiple streams. In our results we found an increase in throughput when using 2 or 4 streams, but using more concurrent streams results in a performance decrease. When increasing the number of streams, the overhead caused by creating and destroying threads also increases, and may have contributed to this performance loss. However, if this were the case, we would observe a much steeper decline in performance for 8 MB messages than for 512 MB messages, as these communications take less time overall, and are thus more easily dominated by threading

[1] DAS-3: http://www.cs.vu.nl/das3/

overhead. We therefore conclude that this overhead is caused by saturation of the local network line. The maximum throughput achieved in these tests is close to the theoretical maximum bandwidth of 10 Gbps. This proves that the MPWide library can efficiently utilize the available bandwidth, if optimal settings are used.

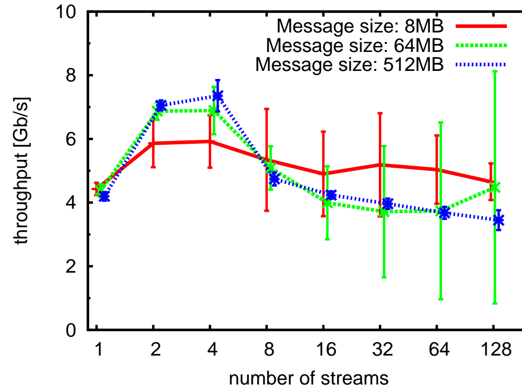

Figure 6.2: Measured throughput in Gbps as a function of the number of communications streams used between two nodes on Huygens. The throughput is given for runs with 1 to 124 threads and message sizes of respectively 8, 64 and 512 MB.

6.4.1.2 National tests

We carried out the national tests over two sites of the DAS-3 Dutch Grid. One site resides at the University of Amsterdam and the other site is located at the Delft University of Technology. The round-trip time of the path between Amsterdam and Delft is 2.1 ms. The results of these tests are found in Fig. 6.3. Although the tests used the regular internet backbone, the fluctuations in our measurements are limited. When exchanging messages of 8 MB size, we obtain the best performance using a single stream, as the use of additional streams results in a lower and more variable performance. This is caused by the fact that message passing performance over multiple streams is limited by the slowest streams. For larger message sizes, however, using a single stream does not result in an optimal performance. Instead, we find that the best results are obtained using 8 streams (for 64 MB) to 32 streams (for 512 MB). Although a high peak performance is obtained when using 64 or more streams, the sustained performance is lower because the excess streams can cause network congestion. The round-trip time of 2.1 ms did not significantly reduce the achieved throughput in our tests.

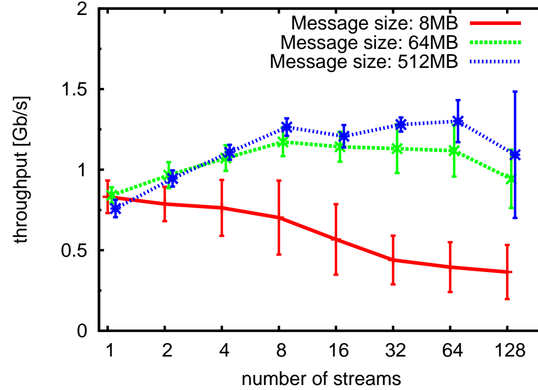

Figure 6.3: Measured throughput in Gbps as a function of the number of communications streams used between the DAS-3 site in Amsterdam and the DAS-3 site in Delft. The throughput is given for runs with 1 to 128 threads and message sizes of respectively 8, 64 and 512 MB.

6.4.1.3 International tests

We show the results of the international tests between Louhi and Huygens, performed in March 2009, in Fig. 6.4. The tests were performed over a shared 10 Gbps network with frequent background network traffic. To minimize the impact of this background traffic, we performed our tests during a quiet period of the day. However, a few of our tests had background interference, causing fluctuations in the measured throughput. When exchanging 8 MB messages, the throughput rate no longer increases once we scale beyond 8 parallel streams. Here, the throughput rate is limited to about 3.5 Gbps due to the high network latency and the small message size. For message sizes of 64 MB and especially 512 MB, the network latency no longer constrains the achieved throughput rate. As a result, we achieved a higher throughput when using more streams. Similar to the national tests, we notice larger fluctuations in performance for larger message sizes. The highest average throughput we achieved was about 4.64 Gbps, which we achieved using 64 streams and a message size of 512 MB.

6.5 Testing performance in a production environment

We originally developed MPWide to manage the long-distance message passing in the CosmoGrid project (see Chapter 4). CosmoGrid is a large-scale cosmological project which aims to perform a dark matter simulation of a cube with sides of 30 Mpc using supercomputers on two continents. In this simulation, we use the cosmological Λ Cold Dark Matter model [48] which defines a constant fraction of the overall energy density for dark energy to model the accelerating expansion of the universe. We apply this

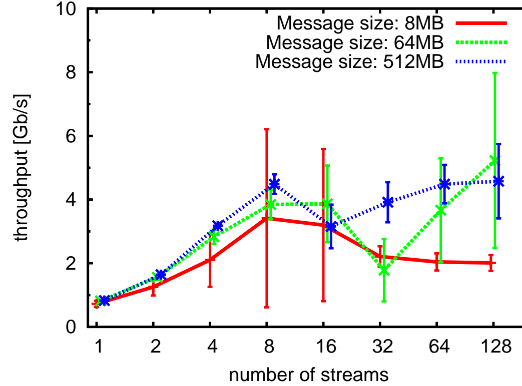

Figure 6.4: Measured throughput in Gbps as a function of the number of communication streams used between Huygens and Louhi. The throughput is given for runs using 1 to 124 streams and message sizes of respectively 8, 64 and 512 MB.

model to simulate the dark matter particles with a parallel tree/particle-mesh N-body integrator, GreeM [65]. This integrator can be run either as a single MPI application, or as multiple MPI applications on different supercomputers. In the latter case, the wide area communications are performed using MPWide. We use GreeM to calculate the dynamical evolution of 2048^3 (~ 8.590 billion) particles over a period of time from redshift $z = 65.35$ to $z = 0$. More information about the parameters used and the scientific rationale can be found in [104].

Before the simulation is launched, the initial condition is decomposed in slices for each site, and in blocks within that slice for each process. Each block contains an equal number of particles but may vary in volume. A simulation process loads one block during startup, and calculates tree and particle mesh force interactions at every step. These force calculations require the exchange of particles with neighboring processes (and sites, see Fig.6.5) as well as the exchange of mesh cells. In addition, a number of smaller communications are performed to balance the load across all processes.

We have used GreeM together with MPWide in a set of test runs, which consist of full-lengths simulations of a limited scale (256^3 particles). Also, we have performed a run across two supercomputers which consists of a limited part of the production simulation described earlier.

6.5.1 Test experiments

We have run three test simulations, of which each one uses a different infrastructure. All runs were carried out over two sites, with 30 calculation processes and one communication process per site. We performed one run on the DAS-3 testbed and one run across the Huygens and Louhi supercomputers. Both infrastructures are described in section

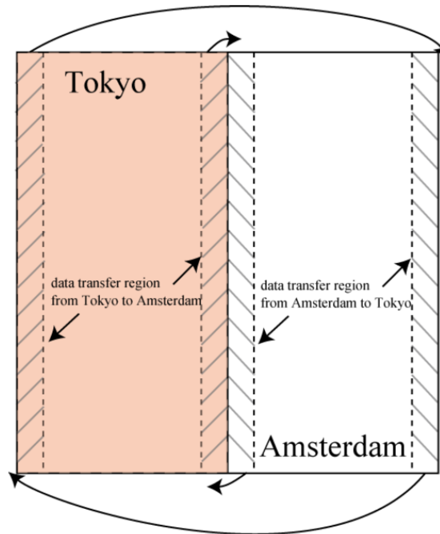

Figure 6.5: Data decomposition overview of the CosmoGrid simulation when run on two supercomputers [104].

6.4, and for both infrastructures we carried out simulations with communication over 1 TCP stream, as the average data volume is only a few MB per communication.

For the third run we have used the Huygens supercomputer in combination with a Cray XT-4 supercomputer located at the Center for Computational Astrophysics in Tokyo, Japan. The Cray XT-4 consists of 740 nodes which run on a quad-core 2.2GHz AMD Opteron and have 8GB RAM each. To exchange data between the sites we reserved and used a 10 Gbps dedicated light path in the GLIF network[117], which has a round trip time of 273 milliseconds. This run was performed prior to the other two runs, using an older version of the code and the library. Unfortunately we were unable to reserve the lightpath for a new test run using our improved setup. For this test we used 64 concurrent TCP streams.

A detailed overview of the communication topology during the simulation can be found in Fig.6.6. Each of the supercomputers has been equipped with one specialized communication node. These nodes are each connected to the high-speed local supercomputer network and are linked together by the 10 Gbps light path. MPWide is used to transfer the locally gathered data to the communication node, forward it to the other site using the light path, and finally to deliver the data to the remote MPI simulation.

6.5.1.1 Results on DAS-3 Dutch grid

The performance results of our test simulation on the DAS-3 can be found in Fig. 6.7. Here we find that the simulation performance is dominated by calculation, with a communication overhead less than 20 percent of the overall wall-clock time throughout the run. As we used regular internet for the wide area communication, our simulation

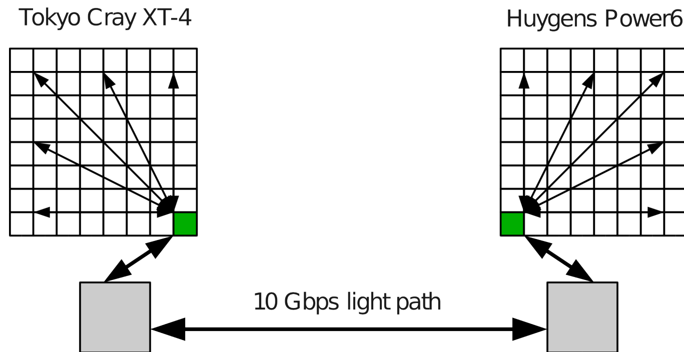

Figure 6.6: Network topology example of the CosmoGrid simulation when run on two supercomputers, one in Amsterdam, the Netherlands and one in Tokyo, Japan. Data transfers within the local supercomputer are performed using MPI (thin arrows), whereas other communications are performed using MPWide (thick arrows). The communication nodes (indicated by the light gray boxes) reside outside of the MPI domains, and therefore use MPWide for all communications. Before the data is transferred to the communication node, it is gathered on a central process on the local supercomputer (indicated by the small gray boxes).

performance is subject to the influence of background network traffic. The two performance dips which can be found around step 1300 and 1350 are most likely caused by incidental increases in background traffic.

6.5.1.2 Results on Amsterdam and Helsinki supercomputers

The performance results of our test simulation between Amsterdam and Helsinki are shown in Fig. 6.8. The obtained performance is similar to that on the DAS-3, although two differences can be noted. First, the calculation time is ~ 25 percent lower due to the superior performance of the supercomputer nodes. Second, although the average communication performance is similar to that observed on the DAS-3, we observe more variability in the communication performance. We are at this point uncertain about the exact nature of this variability. The DEISA network is shared with other institutions, so the presence of background traffic may have decreased our communication performance.

6.5.1.3 Results on Amsterdam and Tokyo supercomputers

The run between Huygens and the Tokyo Cray-XT4 was carried out in October 2008, before any of the other experiments in this paper, and served as a dress rehearsal for both the Tree-PM simulation code and MPWide. The simulated problem was of equal size to the previous simulations and uses the same number of processes. However, we performed the run using an older version of the code and different initial condition files. The performance results of this run can be found in Fig. 6.9. During this test run, the time spent on calculation is roughly constant throughout the run, with a peak

occurring during startup and a few points where snapshots are written. The time spent in communication is generally lower than the calculation time, taking about 7 to 10 seconds per step. However, we also observe a number of communication performance drops. These temporary decreases in performance were almost exclusively caused by single communications stalling for an extended period, which in turn were caused by periods of packet loss.

6.5.2 Production

We have executed a production-sized simulation between Amsterdam and Tokyo to measuring the performance of the code when it is used for production. Based on the results described in Sec. 6.5.1.3, we made a few changes to the network settings before performing the second run. We disabled the TCP memory suppression mode, increased the TCP window sizes, and increased the `sysctl` queue limit for upper-layer processing. Due to the limited length of our network reservation, we were not able to test the effects of modifying each of these settings in detail.

The production-sized run was performed on 752 cores in total. The topology of this run was asymmetric, using 500 cores on Huygens and 250 cores on the Cray for calculation. The full run lasted just under 12 hours, during which we performed 102 simulation steps. The performance results of this run can be found in Fig. 6.10. In this full-scale run, the calculation time dominated the overall performance, and was slightly higher at startup and during steps where snapshots were written. The communication performance is generally more constant than in the small-scale run between Amsterdam and Tokyo , with fewer and less severe performance drops and a slight increase in time after step 30 in the simulation. This increase may have been caused by the TCP buffering sizes, which the local system may change during run-time. Overall, the total communication time per step was between 50 and 60 seconds for most of the simulation, and constituted about one eighth of the total execution time.

6.6 Conclusions and future work

We present MPWide, a communication library to perform message passing between supercomputers. MPWide provides message passing that is intrinsically parallelized, and can be used for high-performance computing across multiple supercomputers. The library allows for customization of individual connections and has a light-weight design, which makes it well-suited for connecting different supercomputer platforms. We have shown results from local and wide area performance tests, and applied MPWide to combine two MPI applications into a very large parallel simulation across several wide area compute infrastructures. During our tests, we reached a sustained throughput of up to 4.64 Gbps over a long-distance 10 Gbps network. In addition, we were able to run an N-body simulation across two continents with 2048^3 particles. During this simulation, about one eighth of the execution time was spent on communications.

Given that the parallel application is sufficiently scalable (which is the case for the N-body integrator used in this work), MPWide can be used to efficiently parallelize

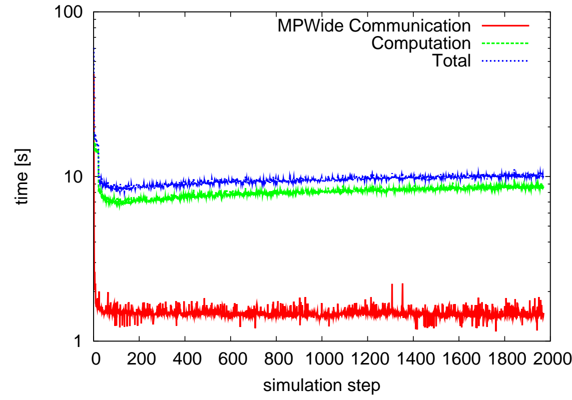

Figure 6.7: Measured wall-clock time spent (in log-scale) on each simulation step for a 256^3 particle test run on the DAS-3 between Amsterdam and Delft. The full-length run was performed using 62 cores, with 30 cores residing on each supercomputer and 2 cores used for communication only. The top dotted line indicates total time spent, the dashed line indicates time spent on calculation and the bottom solid line represents time spent on communication with MPWide.

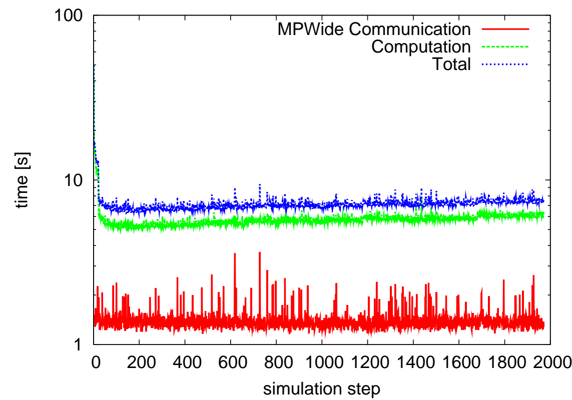

Figure 6.8: Measured wall-clock time spent (in log-scale) on each simulation step for a 256^3 particle test run on the DEISA network between Amsterdam and Helsinki. See Fig. 6.7 for an explanation of the lines.

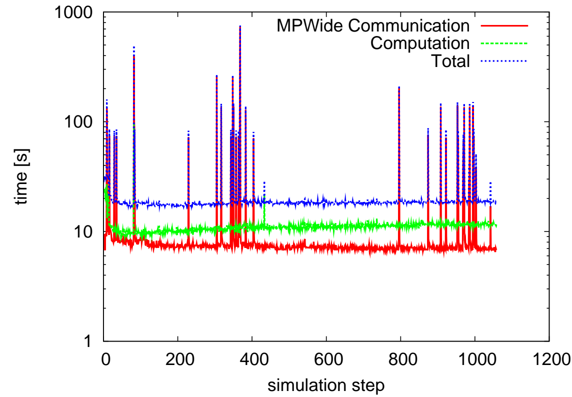

Figure 6.9: Measured wall-clock time spent (in log-scale) on each simulation step for the 256^3 particle test run. See Fig. 6.7 for an explanation of the lines.

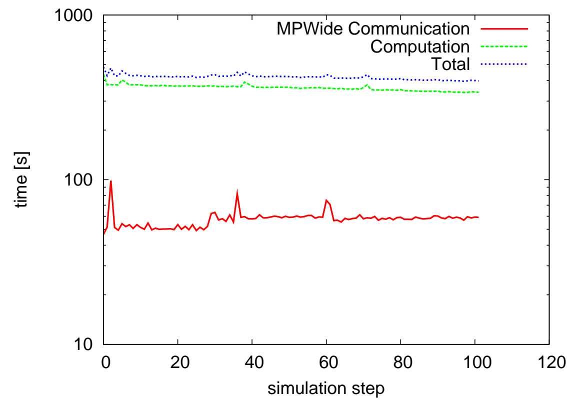

Figure 6.10: Measured wall-clock time (in log-scale) for each simulation step for a partial 2048^3 particle run. The run, which uses some adjusted TCP settings, was performed using 750 cores, with 500 cores used on Huygens and 250 cores used on the Tokyo Cray. An explanation of the lines can be found in the caption of Fig. 6.7.

production applications across multiple supercomputers. Future efforts to improve the usability of MPWide may include the integration with debugging tools and visualization toolkits, the introduction of group communicators and collective operations (similar to `MPI_COMM_WORLD` in MPI implementations), and the addition of automatic deployment mechanisms.