CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Department of Computer Science
Group of Computer Architecture

Diploma Thesis

Analysis and Optimization of the Packet Scheduler in Open MPI

André Lichei
lica@cs.tu-chemnitz.de

Supervisor: Prof. Dr.-Ing. W. Rehm

Adviser: Dipl.-Inf. T. Höfler

Abstract

We compared well known measurement methods for LogGP parameters and discuss their accuracy and network contention. Based on this, a new theoretically exact measurement method that does not saturate the network is derived and explained in detail. The applicability of our method is shown for the low level communication API of Open MPI across several interconnection networks.

Based on the LogGP model, we developed a low overhead packet scheduling algorithm. It can handle different types of interconnects with different characteristics. It is able to produce schedules which are very close to the optimum for both small and large messages. The efficiency of the algorithm for small messages is show for a Open MPI implementation. The implementation uses the LogGP benchmark to obtain the LogGP parameters of the available interconnects and can so adapt to any given system.

The original (PDF) version of this document and the source code created during this work can be found at:

http://archiv.tu-chemnitz.de/pub/2006/0191

Lichei, André:

Analysis and Optimization of the
Packet Scheduler in Open MPI
Diploma Thesis, Chemnitz
University of Technology, 2006

Thesis

It is possible to describe the most important attributes of a interconnect with a few basic parameters of a simple model and predict the performance of this network with sufficient accuracy. LogGP is a model that describes communication networks accurate and is very well suited to give predictions of message delay.

Current available measurement methods could not accurately assess the LogGP parameters of a communicating network in short time or without saturating the network. It is possible to implement a measurement method which can assess the LogGP parameter of a given network in short time.

In commonly used systems the delay suffered while a packet is transported through a network is far greater than the sum of the other delays. The capacity of the network is far smaller than the capacity of the local system.

It is possible to reduce the time a set of messages needs to get transmitted by sending them through several interconnects in parallel.

It is possible to improve the performance of a communication library or a application through scheduling of small messages across several available communication networks.

It is possible to develop a scheduling algorithm for small messages which is independent from the used hardware and can still work efficiently. All necessary informations can be obtain while runtime without creating a large overhead.

Table of Content

1.Introduction....7
2.Models of network performance....8
2.1Characterisation of available architectures....8
2.1.1Factors which will Affect Network Performance....8
2.2Models....9
2.2.1Queuing Networks....9
2.2.2Hockney model....9
2.2.3C3....10
2.2.4BSP....10
2.2.5LogP....10
2.2.6PlogP....10
2.2.7LogGP....10
2.2.8LogP with stalling....11
2.2.9LogGPS....12
2.2.10LoPC....12
2.2.11LoGPC....12
2.2.12Evaluating Models....12
3.Measurement of the LogGP Parameter....15
3.1Related Work....15
3.1.1LogP Performance Assessment of Fast Network Interfaces....15
3.1.2Cross-platform Analysis of Fast Messages for Myrinet....15
3.1.3Fast Measurement of LogP Parameters for Message Passing Platforms....15
3.1.4An Evaluation of Current High-Performance Networks....17
3.2Theoretical Approach to a New Method....17
3.2.1The Parametrized Round Trip Time....17
3.2.2The Parameterized Round Trip Time under LogGP....18
3.2.3Computation of the LogGP Parameter....19
3.3Statistical Basics Necessary for our Approach....20
3.3.1Why do we have to use Statistical Methods....20
3.3.2Interpreting Measured Values....20
3.3.3Error Distribution....21
3.3.4Linear Regression....21
3.4Implementation....22
3.4.1Characteristics of the Existing System....22
3.4.2Implementing the PRTT Micro Benchmark....22
3.4.3Implementation of the LogP Benchmark....24
3.5Results....24
3.5.1Used Systems:....24
3.5.2The PRTT Values....24
3.5.3The resulting LogGP parameters....26
3.6Conclusion and future work....28
4.Scheduling....29
4.1Taxonomy....29
4.2Current Scheduling....29
4.3Evaluation Scheduling Strategies....30
4.3.1Optimal Scheduling....30
4.3.2Real Scheduling....31
4.3.3Evaluation of the Current Scheduling Algorithm....31
4.3.4Some basic scheduling algorithms....32

4.4Developing a Scheduling Algorithm Adapted for Open MPI....33
4.4.1Simple scheduling....33
4.4.2Scheduling for Packets of Different Size....34
4.4.3Scheduling Aware of Pauses Between Consecutive Messages....35
4.4.4Scheduling Aware of the Latency....35
4.4.5Scheduling Aware of Parallelism of Overhead and Gap....36
4.4.6Scheduling for More than Two Nodes....36
4.4.7Reducing Runtime of the Scheduling Algorithm....37
4.4.8The Final Algorithm....37
4.5Further improvements....38
4.6Classification and Measurement of the LogGP Parameter....38
5.Implementation....40
5.1Open MPI Overview....40
5.2MCA....40
5.3The Layers of Open MPI Responsible for Communication....42
5.4Choosing a Framework....42
5.5Current Implementation of the BML Framework....43
5.5.1Opening and Closing the R2 Component....43
5.5.2Initializing the R2 Modules....43
5.5.3Adding / Destroying a Processes....44
5.5.4Adding / Deleting BTL Modules....45
5.5.5Registering a Callback Function....45
5.5.6Finalizing the Module....46
5.5.7Methods of the mca_bml_base_btl_array_t Class....46
5.5.8Functions Provided by the Framework for Communication....46
5.5.9Interactions between other Frameworks and the BML while Communicating....47
5.6Implementing Scheduling in the BML Framework....47
5.6.1Global Access to the Scheduling Information....48
5.6.2Involving the BML Component in Communication....48
5.6.3Obtaining Descriptors....49
5.6.4Sending Data....50
5.6.5Receiving Data....50
5.6.6Using RDMA....50
5.7Performance Analysis of our Design....50
5.7.1Use of Function Pointers....50
5.7.2Memory Segments....51
5.8Results....51
5.8.1The Microbenchmark....51
5.8.2The Used Test Systems....52
5.8.3The Performance with one Network....53
5.8.4The Performance Using two Identical Networks....54
5.8.5The performance using two Different Networks....55
5.8.6The Benefit of Scheduling....56
5.9Overcoming the Bandwidth Limitations for Large Messages....60
6.Conclusion and Further Works....60
7.Appendix....1
7.1References....1
7.2List of Figures....3
7.3List of tables....4
7.4List of Abbreviations....4
7.5Acknowledgements....4
7.6List of Trademarks....5

1. Introduction

Motivation:
In the past couple of years, much work was done to develop efficient high performance environments by Cluster Computing. Commodity PC's which provides high computational power and network interconnects which provides low latency and high bandwidth form the basis of such High Performance Cluster (HPCs). In this area, Message Passing Interface (MPI) ([MPI95];[MPI97]) has become the de facto standard of programming applications.
In such Cluster systems the interconnections networks have become a critical component of the computing technology with a direct impact on the design, architecture, and use. Especially for clusters build with symmetric multiprocessor systems (SMP) machines the network bandwidth of the interconnect can become the performance bottleneck. A natural solution is to use several interconnects in parallel and thus enhancing available bandwidth. Support for several interconnects can be implemented in available MPI libraries and so make high bandwidth available to a width field of applications. Recently developed systems like the Open MPI [GFB04] library support the use of several interconnects and distribute messages across several interconnects. Much work was done to achieve efficient distribution of large messages and their fragments but, to the best of our knowledge, very little attention has been given so far to the distribution of small messages. When large clusters with a high number of nodes are used the efficiency of collective operations like the MPI_Barrier depends on a fast transmission of a large number of small messages. Collective operations are fundamental for parallel applications so the efficient transmission of small messages becomes a crucial issue. Therefore it shall be determined how small messages can be distributed across several different interconnects and a scheduling algorithm should be implemented in the Open MPI library.

Task Formulation
The objective of this diploma thesis is to develop a prototype for a packet scheduler. It shall be shown experimental whether scheduling of small messages can lead to higher performance of a communication library or not. If so a effective scheduling algorithm should be developed and implemented.
The work will be based upon a current version of Open MPI. The existing packet scheduler shall be extended and optimized. A solution should be developed that allows a efficient scheduling of small messages across several interconnects with different characteristics. Thereby it is important that scheduling is be done transparent to the application which uses Open MPI. To guarantee a high portability the scheduling algorithm has to be hardware independent and the characteristics of each interconnect shall be assessed during run time. The characteristics shall be used by the algorithm to adapt to any given situation and provide optimal scheduling for every system. Therefore a accurate and hardware independent modelling of interconnects have to be found. The model must provide accurate prediction of network performance yet have to be simple enough that it can be used during run time without introducing much overhead.
A assessment method which could be used during runtime to assess the parameter for the chosen model has to be found or developed. It is important that assessment can be done in reasonable time and do not significantly delay the execution of applications which are using Open MPI.
As result a new component of the Modular Component Architecture (MCA [SL04]) shall be developed and shall enable the user to choose whether he wants to use the new introduced algorithm or not.

2. Models of network performance

The aim of the to develop algorithm is to achieve an efficient distribution of communication requests across several communication interfaces. To achieve such a distribution the performance of each interface and the underlying communication network has to be determined. To generate a program which is usable on a variety of systems the performance has to be determined at runtime. Only the user can provide detailed information about the structure and technologies of the communication networks. This information may be difficult to obtain by the user so we assume this information as unknown, so the procedures used to determine the performance must be independent from the used networks. Thus a model has to be found which satisfies the following requests.

- the model has do be architecture independent
- it has to capture the most important parameters and must allow an accurate performance prediction of the used communication network
- yet has to be simple enough to allow the development of simple and fast scheduling strategies
- methods for assessing/measure the model parameters for each network at runtime have to be available

2.1 Characterisation of available architectures

In this subsection the architectures which are being reviewed in this work will be characterized. A classification of parallel computer architectures gives [War98]. Without any further explanation it is stated that this work will concentrate on

- boundoir machines
- using the architecture model of distributed memory precisely message passing
- focus on clusters of standard PCs

2.1.1 Factors which will Affect Network Performance

[War98] distinguishes between 4 main factors

1) network delay
2) access point delay
3) operating system delay
4) API delay

A network delay means the time a network needs to process a given communication request. First a route between communication partners has to be found. This delay occurs on every involved node. So the network structure and the used routing algorithm plays an important role. Then the data has to be transported. Obviously the amount of data is important and depending on the used routing strategy (Store-and-Forward[JMY89]; Wormhole[NK93]; Virtual-Cut-Through[PJC96] and Circuit-Switching[PJC96]) the number of nodes involved in the communication can play a role. For examples of routing algorithms see [Sch98], [YD98].

A access point delay means the time data needs to reach the network interface. It is highly dependent from the point where the network interface is integrated in the system. [War98] distinguishes between 4 possibilities with different performance.

- integrated in the CPU
- connected to the processor bus

- connected to the memory bus
- connected to the I/O bus

In standard PCs the network interface is commonly connected to the I/O bus but lately systems with network interfaces integrated in the CPU are available.
In many systems the network interface is manage by the operating system. A operating system delay means the time the system needs to process a system call. This time may be a significant delay. So it was observed that protocols which divide large messages in smaller packets at the operating system layer outperform protocols which do not provide such service and so force the user to do the division in user space. For example see [RHR06].
This work bases on the MPI implementation Open MPI [GFB04]. More precisely a new improved BML component is designed. The BML uses the underlying BTL components for communication. So the delay communication suffers through computation within the BTL layer is determined through the available and used BTL modules and can be referred to as API delay.

2.1.1.1 Criteria for Parallelisation

When using different communication networks/interfaces in parallel, only the network delay may be reduced for a set of packets. So to achieve speedup when using different networks in parallel a fundamental requirement is:

> In commonly used systems the delay suffered while a packet is transported through a network is far greater than the sum of the other delays. The capacity of the network is far smaller than the capacity of the local system.

2.2 Models

The description above outlines the difficulty with trying to describe an entire communication system accurately. Thus models have to be found which describe only the important attributes. Thus made it easier to evaluate the performance of each available network. This section deals with several well known models. Both models for network description and for parallel programming are discussed. Even though models for parallel programming are not intended for modeling network behaviour, they too have to use quite simple models for describing network performance.
Each model is described very briefly, yet a reference to the original publication is given for additional details. Models using similar concepts of modeling network behaviour are associated in several classes, the most suited class is chosen and the models out of this class are compared in more detail. As result the most suitable model is chosen.

2.2.1 Queuing Networks

The theory underlying queues is highly developed and properties can be derived quite generally. Thus queues are central in many models of communication systems. A main advantage is that queues can model contention. Any queue is defined in 3 stages: one an arrival process determining when packets arrive and possibly what their characteristics are, two a buffer where packets wait for serving and three a server serving a queue. Most of the attributes are described with random distributions.
Complex systems are modelled with queuing networks consisting of several queues. These models allow statistical description of the system in equilibrium. For a detailed discussion about queues, queuing networks, their use and limitations see [HP93].

2.2.2 Hockney model

In the Hockney model [H94] it is assumed that the time to send a message of size m can be

calculated through $\alpha + \beta * m$, where α is the latency for each message, and β is the transfer time per byte or the reciprocal of the network bandwidth.

2.2.3 C3

The C3 model was proposed by Hambrush et al in 1994 [HK94]. It was developed for coarse grained supercomputers, and works by dividing algorithms into several supersteps. Each superstep consists of an computation step followed by a communication step. For determining the length of the communication step, the length of every message transmission is calculated. Every message is split into fixed size packets. Communication set up costs, latency and bandwidth are used in different formulas depending on the routing type (store and forward [JMY89]; wormehole [NK93]) and whether blocking or nonblocking sends, receives are used. Additionally by using the number of processor pairs communicating in one superstep the contention is estimated.

2.2.4 BSP

The Bulk Synchronous Parallel (BSP) model was introduced by Valiant in 1990 [Val90]. This model too divides the algorithm into several consecutive supersteps each consisting of a computation step and a communication step. In each communication step each processor may send h messages and receive h messages. A cost of g*h is charged for the communication whereas g is a bandwidth parameter.

2.2.5 LogP

Culler et al. proposed in 1993 the LogP model [CKP+93]. The model is intended to be used for coarse grained machines representing a set of complete computers each consisting of one or more processors, cache, main memory connected by a communication network.
The four main parameters are:

- L(latency): an upper bound on the delay which a packet experiences while traveling from its source interconnect interface to its target interconnect interface.
- o(overhead): is the time a processor is involved in sending or receiving a packet.
- g(gap): is the minimum time interval between consecutive message transmission or reception. Note: for each further packet after the first one o is covered by g.
- P: the number of processor/memory module

Figure 2.2.7.1 provides a example for the LogP scheme.
In addition to the Parameters following assumptions are made:

- L/g represents the number of messages which can be in transmission on the network from any or towards any processor in parallel. Every processor which tries to insert more packets will stall until packets receive their destination.
- All packets have a fixed size. Larger messages will be divided into several packets.

2.2.6 PlogP

In [BKV00] parameterized LogP was proposed by Khielmann et al. As in LogP, L is used to describe the end-to-end latency from one process to another. Whereas in LogP o was used as parameter for both the sending and receiving site, PlogP distinguishes between os at the sending site and or at the receiving site. Furthermore os(m), or(m), g(m) are defined as functions of the message size m. Furthermore it is assumed that the network needs the whole g to send each message, including the first. So a message is received after L+g. See Figure 2.2.7.1 for a example.

2.2.7 LogGP

The LogGP model was proposed by Alexandrov et al. in 1995 [AIS95] as an improvement of the LogP model. A new parameter G as “per byte gap” was introduced. Thus allowing allgorithm to benefit from modern interconnect technologies with special support for long messages. g of the original LogP now only functions as a minimum time interval between consecutive messages. Wheras g+G*m is the time interval between two messages of size m. The scheme is illustrated in Figure 2.2.7.1.

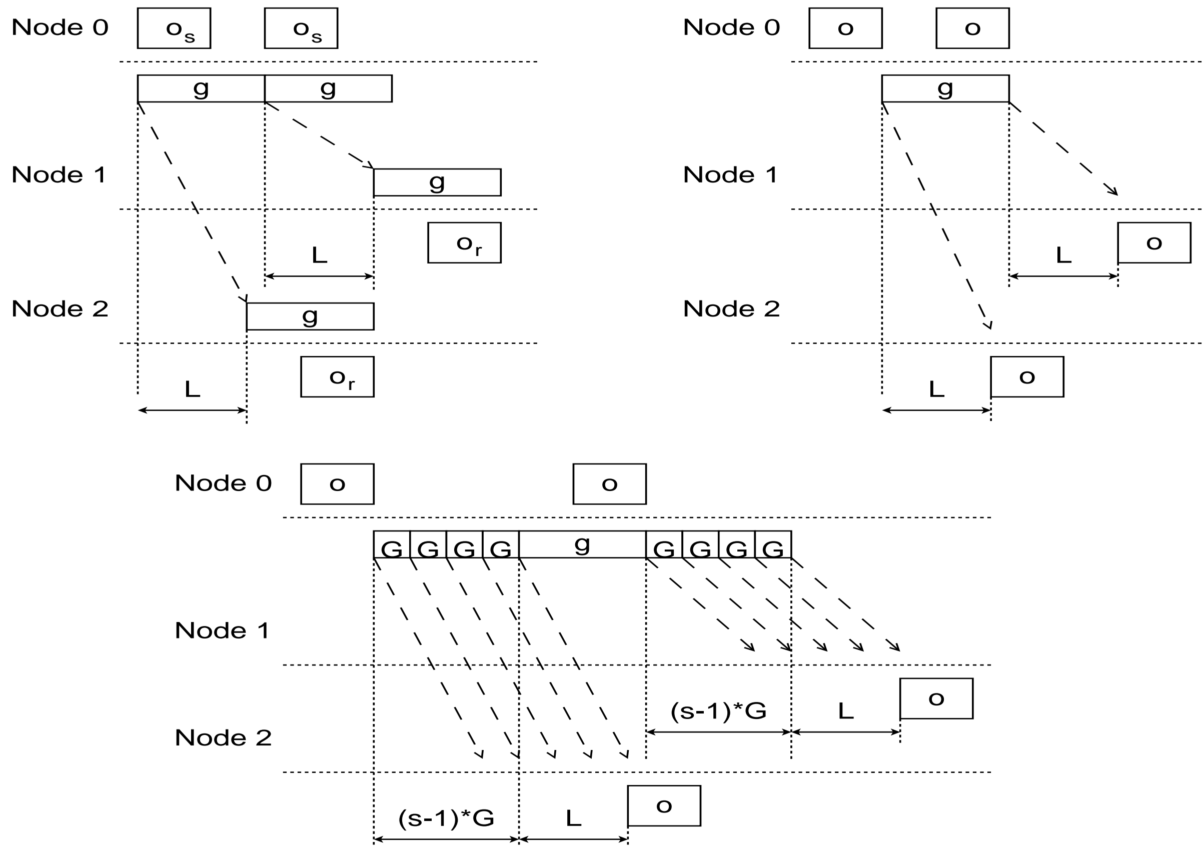


Figure 2.2.7.2: Transmission scheme to two nodes in: upper left PLogP; upper right LogP; down LogGP (5 bytes)
Figure 2.2.7.1: Transmission scheme to to nodes in: upper left PLogP; upper right LogGP; down LogGP

2.2.8 LogP with stalling

In 2005 Bilardi et al. proposed an extension of the LogP model [BHP05]. LogP was intended to reward the algorithm design which prevents contention in the network. It was difficult to estimate however, the runtime of algorithms which may lead to stalling because of network contention. So LogP was extended by a new parameter δ representing the time needed to inform a processor that it has to stall. Thus enhancing the power of the model, which can now be used to design algorithms that may stall and make a proper runtime estimation.

2.2.9 LogGPS

The LogGPS [IFH01]; an extension of the LogGP model, was introduced in 2001 by Ino et al. It covers additional delay occurring when using several MPI ([MPI95],[MPI97]) libraries due to the rendezvous protocol used for long message transfer. An additional parameter S is the threshold for message length, above which synchronisation occurs.

2.2.10 LoPC

In 1997 Frank et al. proposed the LoPC model [FAV97]. It deals with contention for message processing resources in the processor nodes. Stochastic theory is used to model both application and architecture behaviour, thus allowing to predict the impact of contention on the runtime of an application.

2.2.11 LoGPC

Moritz et al. proposed in 2001 the LoGPC model [MF01]. The impact of network contention and network interface DMA behaviour is incorporated in the LogP Model. The model was developed for k-ary n-cube networks using wormhole routing [NK93]. As in LoPC stochastic theory provides the basics for modelling application and architecture behaviour.

2.2.12 Evaluating Models

2.2.12.1 Classification of Models

A main difference between these models is the amount of information used to calculated network delay. The application can be seen as a white box e. g. at least statistical information about the message input rate at each node is available and/or the number of node pairs in communication at each point in time is available. Or it can be seen as black box e. g. the application is expected to produce regularly messages at each node, but nothing more is known. Available network information can be similarly classified. The network can appear as a white box meaning information about the structure (network topographic, type and performance of switching nodes) and/or the used routing protocols are available. Or it can appear as black box meaning only information about the expected delay of an single message submitted to the network is available. In table 2.2.12.1 the Models are assigned to the matching class by comparing the available model descriptions.

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">Network specification</td></tr>
<tr><td>white box</td><td>black box</td></tr>
<tr><td rowspan="2">Application specification</td><td>white box</td><td>Queuing Networks
C3
LoPC
LoGPC</td><td>BSP</td></tr>
<tr><td>black box</td><td></td><td>Hockney model
LogP
PLogP
LogGP
LogP with stalling
LogGPS</td></tr>
</table>

Table 2.2.12.1: Classification of network models

2.2.12.2 Choosing a Class

When using a model the required information has to be determined. Since the to developing algorithm have to be architecture independent, no information is as a priori available. Thus meaning all required information have to be measured. Information about the structure of the used networks can only be gathered by complex indirected measurement. Thus increasing the error made while measuring and so making the gained increase of accuracy questionable. (For a discussion about errors made while measuring see subsection 3.3, for details about error distribution for indirect measurement see [Pap99]) So the increased effort while measuring seems not to be justified, especially since the impact of the network structure on model parameter seems not to be to important. (For a discussions about the impact of the network structure see [CKP+93] Chapter 5 “Matching the Model to Real Machines”)
Measuring application behaviour suffers under similar difficulties. The application behaviour can only be measured at runtime. In order to make the measured results available throughout the entire system additional communication is required. On the one hand the measurement may increase latency or at least may affect application performance on the other hand distributing the information is narrowing the available network performance which is already recognized as bottleneck. So gathering the information for white box models in a sense of application behaviour seems does not seem to be reasonable.
With such boundaries no model is left witch can model contention. This is not seen as a disadvantage. Whenever packet transmission does not follow the modelled behaviour contention can be estimated as cause. This provides a useful method for contention detection.

Note: We are aware, that the above stated problems may be solvable. (For example distributing information through the system may be done by backpacking information on messages of the application) But exploring such methods requires extensive additional work. Since this work is intended to serve as a starting point, it was decided to concentrate on the most promising class of models, the “black box / black box” approach.

2.2.12.3 Choosing a Model

In this subsection the remaining models are compared. As a result the most suitable model is chosen.

Hockney model
The Hockney model is quite simple and so allows the design of fast scheduling algorithms. However other models can provide better accuracy.

LogP
The LogP model provides better accuracy than the Hockney model, yet is simple enough to allow fast scheduling algorithms. Especially the ability to distinguish between the delay the processor is involved in and the delay caused by the communication network makes this model interesting. Unfortunately LogP is only accurate when using packets of a single size. So it can not be used here.

LogP with stalling
During this work interesting extensions to the LogP model were made. However these extensions only influence application design and not network modeling and so provides no improvement for this work.

LogGPS
LogGPS increase the accuracy only when using a MPI communication library. Since we are using low level communication protocols LogGPS do not gain further improvements.

PLogP
PLogP deals with different packet sizes. Thus makes it more accurate than LogP. The main goal of the developer of PLogP was to provide a model with easy to assess parameters. So most of the extensions simplify the measurement, but complicate usability and decline accuracy. So using PLogP complicates scheduling algorithm design.

LogGP
LogGP too deals with different packet sizes, but is not as complex as PLogP. Additional LogGP provides bandwidth information depending on packet size. So it helps finding packet sizes to divide large messages in. At all LogGP seems to be the most promising choice.

As a result LogGP is chosen to serve as base for further investigations.

3. Measurement of the LogGP Parameter

In this chapter we compare different approaches to measure the LogGP parameters and as a result develop our own approach. We explain the theoretical basics and show in the result section that our approach is valid.

3.1 Related Work

We analysed four available benchmarks. One for LogP and one for PLogP, the two others used their own models which are slightly different to LogP. We show strengths and weaknesses and then decide based on the results not to use any of them.

3.1.1 LogP Performance Assessment of Fast Network Interfaces

In [CLMY96] Culler et al. proposes a method of measuring the LogP parameter of existing high performance networks. The basic idea is to measure the time active message send calls needs to finish. He differentiates between o_s on the sender side and o_r on the receiver side which complicates the model slightly. The time one call needs to complete is considered to be the o_s parameter of the LogP model. When sending several messages in a row, the time one call needs to finish is not increasing, because messages are buffered in the system. When the buffer is filled and the network is saturated each new message has to wait for g until it can be processed. So the time one send call needs to finish corresponds to g when the network is saturated. The L parameter is then calculated with the round trip time of an single message. (For details see [CLMY96])
We came to the conclusion that we should not adopt this method for measuring the LogGP parameters in Open MPI. The measurements of o_s could be problematic on modern architectures, because they tend to copy the message to a temporary buffer and process it later (e.g. TCP). The second problem we see, is that the network has to be saturated before we could measure g. Since it may be that the operating system is buffering messages it may be that we have to fill great areas of the main memory before this saturation occurs. This means we have to send a very large number of messages when we want to measure g for small sized messages e.g. one byte messages. Accurate measurement is only possible with a perfect working flow control scheme of the network. As it has to prevent a performance loss in the network merely because of saturation. So we will try to prevent saturation when possible to reduce the number of error sources.

3.1.2 Cross-platform Analysis of Fast Messages for Myrinet

In [ILM98] Iannello et. al. presents LogP for different systems using Myrinet®/GM. He uses similar techniques to assess the LogP parameters like Culler et. al.

3.1.3 Fast Measurement of LogP Parameters for Message Passing Platforms

In [BKV00] Kielmann et. al. proposes an method for measuring the PLogP parameter of high performance networks. They also distinguish between o_s for the overhead on the sender side and o_r on the receiver side. The time a single send call takes to finish is considered to be o_s. To measure o_r the receiver has to wait until it can guarantee that the message has arrived. Then the time the receive call needs to finish is considered to be o_r. For measuring the g and the L parameter the round trip time of packages different sizes is measured. Round trip time means the time needed to send one package of a certain size from the sender to the receiver and an acknowledgement back to the sender. Using round trip times makes sure that buffering does not influence the result, since it

ensures that the package has reached the receiver in the measured time. Thus saturation of the network should not be necessary. g for 0 byte messages is measured by sending n messages in a row. Then the receiver sends the acknowledgement. The time divided by n is considered to be g(0). The author describe that the network has to be saturated before g could be measured. So n must be high enough. No more explanation is given and it is unclear why saturation is needed. (As stated above filling the buffer should not be necessary.) Furthermore it is unclear why the round trip time of n packages divided by n should be equal to g. If we use the PLogP model to describe the round trip time for a 0 byte package we get the following (considering that the sender and the receiver are equal nodes and the network performs in both directions similarly.)

$$RTT(0) = (L + g*n) + (g + L)$$
$$= 2*L + (n+1)*g$$
$$= 2*L + g + n*g$$

If we divide this by n we get

$$\frac{RTT(0)}{n} = \frac{2*L+g}{n} + \frac{n*g}{n}$$
$$= \frac{2*L+g}{n} + g$$

If we converge n to ∞ we get

$$\lim_{n\to\infty} \frac{RTT(0)}{n} = \lim_{n\to\infty} \frac{2*L+g}{n} + g$$
$$= g$$

So if a high n is used we get g with a small error. But this n may be quite high. For an example we use the results we got while measuring the LogGP parameter of an TCP connection across Gigabit Ethernet. (For details of the used measuring method see section 3.2.1)

L=76,5μs

g=0,783μs

If we want the error to be smaller than 1% we have to use an n for which the following inequality holds:

$$0{,}01*g > \frac{2*L+g}{n}$$
$$n > \frac{2*L+g}{0{,}01*g}$$
$$n > \frac{2*76{,}5+0{,}783}{0{,}00783} > 19640$$

To compute g for other sized messages Kielmann uses g(0) and the round trip time (RTT).

$$g(m) = RTT(m) - RTT(0) + g(0)$$

Than L is computed with

$$L = \frac{(RTT(0) - 2g(0))}{2}$$

In conclusion we decided not to uses this approach unchanged. Using the round trip time and so avoiding saturation of the network is promising. But the measurement of the o parameter may lead to false results. TCP for example is computing after the send call finished, and is computing when a message arrives even when no matching receive request was posted. It is also difficult to get good

results when measuring g(0), because of the large number of messages that have to be used. The approach will indeed saturate the network. This we want to avoid if possible.

3.1.4 An Evaluation of Current High-Performance Networks

The latest work was proposed by Bell et al. in [BBC03]. The parameter o_s is measured with a delay between message sends. This delay d is adjusted until d+o fits g+(s-1)G exactly. Now, o_s is computed via g+(s-1)G-d, which relies on the correctness of g and G. The method to assess o_r is similar to the method in [CLMY96], but Bell delays the transmission of the answer on the receiver side. He uses a similar technique as Kielmann to measure g. L can not be measured because modern networks tend to start the message transmission before the CPU is done (L is started before o ends). Bell et al. introduce the end to end latency (EEL) which denotes the RTT for a very small packet. Measuring o_s with a adjusted d requires multiple measurement steps to adjust the correct d. The method to measure g suffers from the same problem that it has to send a huge number of packets (n) to get an accurate measurement and the network is effectively flooded. So we do not use this approach, since it does not prevent saturation of the network. Furthermore the method used to assess o_s is far too expensive.

3.2 Theoretical Approach to a New Method

In this subsection we present a micro benchmark and show how to calculate the LogGP parameters out of the results of the micro benchmark.

3.2.1 The Parametrized Round Trip Time

It is not possible to assess the LogGP parameter directly. If we want to determine the parameter we have to measure other direct accessible parameters and we have to compute the LogGP parameter. Commonly used benchmarks uses the round trip time (RTT) as a possibility to evaluate the performance of an interconnect. Since the round trip is used successfully in a wide field of benchmarks and is easy to assess, it seems to be reasonable to try and establish our measurement method upon the round trip time.
Commonly the round trip time is understood to be the time needed to send on packed from one node to another an then immediately sending it back. The time needed depends on the packet size (s), and so the round trip time is a function of the size:RTT(s).
In future we will use request to address the packet sent from the first node, and reply to address the answer the second node sent. We want to expand the term round trip time, and want to use two more parameter:
number of packets (n) (like the ping ping scheme mentioned in [Pal00])
delay time (d)
So we get the parametrized round trip time $PRTT_{n,d}(s)$.
The number of packets gives the number of packets of size s sent in the request and must be at least one. The reply will consist of one packet of size s in any case. To send a reply of size s is not necessary in any case. A reply of a 0 byte packet would be sufficient in some cases. We use the size s packet because then the half time of PRTT1,0(s) is the time one packets needs to travel from the sender to the receiver, and we want to use the same scheme for all cases. The delay time is only used when the number of packets is greater then one. It gives the time the sender waits between sending two consecutive packets of one request. whereas it must assure that the processor is computing throughout the whole waiting time. As a result the common round trip time is a special case of the parametrized round trip time RTT(s) = $PRTT_{1,0}(s)$.
The $PRTT_{n,d}(s)$ is illustrated by the following pseudo code. (We assume that the time is measured in processor cycles.)

Start Timer
  send(s bytes)
  repeat (n-1) times
    compute d cycles
    send(s bytes)
  recv()
Stop Timer

3.2.2 The Parameterized Round Trip Time under LogGP

We assume that both nodes used to measure the parametrized round trip time are identical and that the interconnect performs equally in both directions. So we can predict the parametrized round trip time with the help of the LogGP model.Considering that n=1 and d=0 we get:

$$RTT_{1,0}(s) = 2*(o_s + L + o_r + (s-1)*G)$$

Equation 3.2.2.1

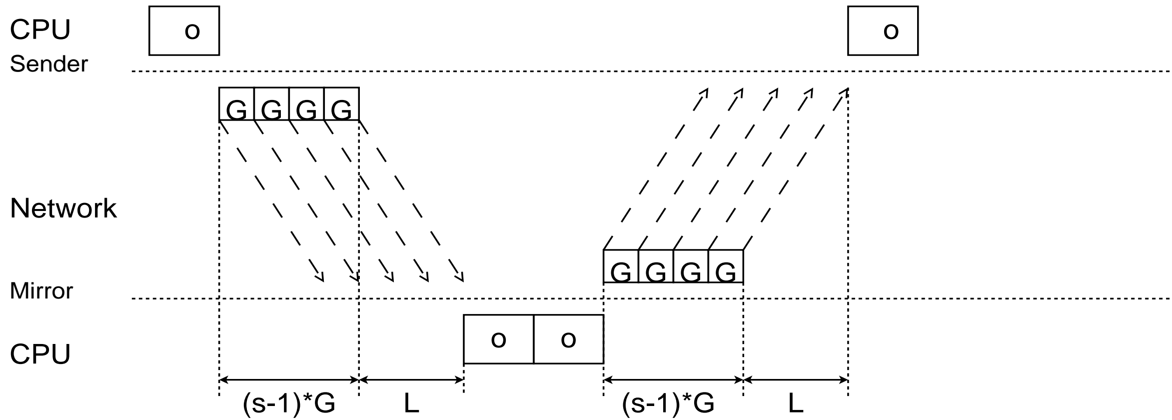


Figure 3.2.2.1: PRTT for 1 5 byte packet: green the total time for one transmission

Considering d=0 we get:

$$\begin{aligned} RTT_{n,0}(s) &= 2*(o_s + L + o_r + (s-1)*G) + (n-1)*max(o_s, g + (s-1)*G) \\ &= RTT_{1,0}(s) + (n-1)*max(o_s, g + (s-1)*G) \end{aligned}$$

Equation 3.2.2.2

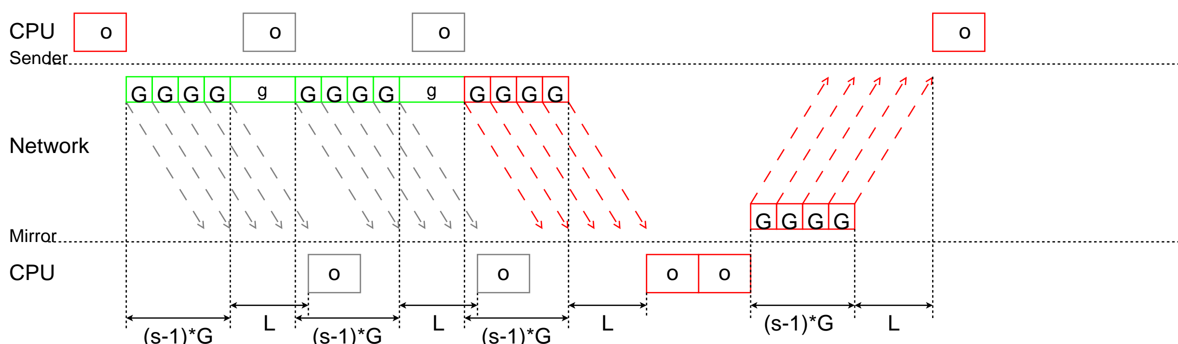


*Figure 3.2.2.2: PRTT for 3 5 byte packets without delay: red the RTT for one packet; green the time for g+(5-1)*G; gray the part without influence on the result time*

And the general case:

$$RTT_{n,d}(s) = 2*(o_s+L+o_r+(s-1)*G)+(n-1)*max(o_s+d, g+G*(s-1)) = RTT_{1,0}(s)+(n-1)*max(o_s+d, g+G*(s-1))$$

Equation 3.2.2.3

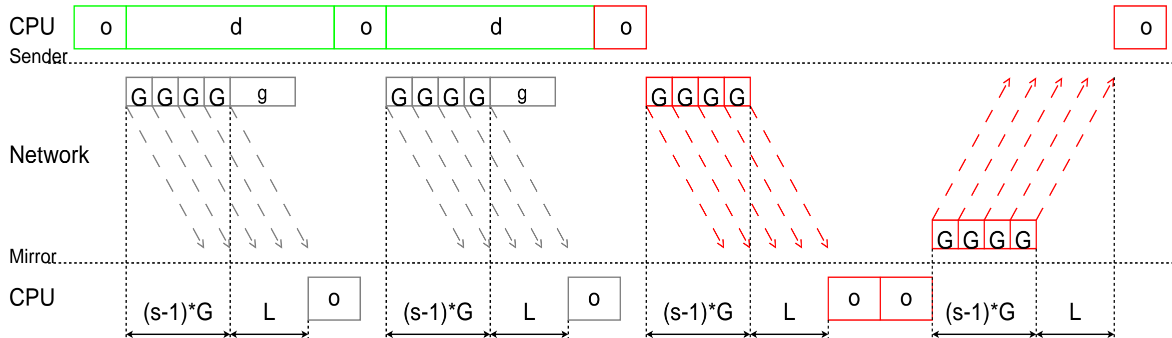


Figure 3.2.2.3: The PRTT for 3 5 byte packets with a delay of d: red the RRT for one packet; green the time of o+d; gray parts that do not influence the result

3.2.3 Computation of the LogGP Parameter

For the sake of simplicity we assume that $o<g$. Then we can rewrite the given equations (The right part of the equations is green in the figures 3.2.2.1, 3.2.2.2, 3.2.2.3).
We rewrite Equation 3.2.2.1 to

$$\frac{RTT_{1,0}(s)}{2}-2*o=L+G*(s-1) ,$$

Equation 3.2.3.1

Equation 3.2.2.2 to

$$\frac{RTT_{n,0}(s)-RTT_{1,0}(s)}{n-1}=g+G*(s-1)$$, and

Equation 3.2.3.2

Equation 3.2.2.3 to

$$\frac{RTT_{n,d}(s)-RTT_{1,0}(s)}{n-1}=max(o+d, g+(s-1)*G) .$$

Equation 3.2.3.3

3.2.3.1 Assessing g,G

Measuring $PRTT_{1,0}(s)$ and $PRTT_{n,0}(s)$ for one fixed n and several s, enable us to fit a linear function f(x)=y=a*(x-1)+b which represents the Equation 3.2.3.2. We draw the size at the x-axis and the results of Equation 3.2.3.2 at the y-axis. The parameter of the linear function are the searched g and G parameter of the LogGP model f(x)=y=G*(x-1)+g. How to fit this function properly is shown in

3.2.3.2 Assessing o

For a large enough d exist an s_i so that $\frac{PRTT_{n,d}(s)-PRTT_{1,0}(s)}{n-1}=o+d \; \forall \; s<s_i$

As result we can compute o when we choose an appropriate d and fixed size s.

3.2.3.3 Assessing L

Similar to we could fit a linear function to Equation 3.2.3.1. The intercept of this function represents L (f(x)=G*(x-1) + L).

3.3 Statistical Basics Necessary for our Approach

In this subsection we present some necessary statistical approaches. We show in brief why they are necessary and how to use them. The three main topics are how to improve the quality of measured values, error distribution in computations with measured values and how to fit a linear function to coordinates we obtained throughout the measurement

3.3.1 Why do we have to use Statistical Methods

Computer systems as we use them are deterministic. So it should be sufficient to measure the needed PRTT values once and compute the LogGP parameters. Unfortunately, a modern computer system is complex. It consists of a large number of subsystems which influence each other. So the time needed to transmit a packet varies. Statistical approaches help to interpret such influenced measurements.

3.3.2 Interpreting Measured Values

Physical science faces similar problems to ours. When measuring is done many small influences occur that could not be controlled nor predicted. A great number of approaches are available to minimize the effect such influences have on measurement. We can not cover this topic in depth. For detailed information see [Pap99]:chapter IV. The basic idea is that the number of influences is very high and the effect each has is small. The value which should be measured is influenced in an orderless manner. This mean that the result can become greater or smaller with the same probability. So in conclusion the measured value is a continuous random variable which is normally distributed. Now the measurement is done several times. The results are considered as a sample. The parameter of the distribution of the real value is estimated by the parameters of the sample. Normally the mean value of the sample is used to estimate the mean of the distribution of the value and the standard deviation of the sample is used to estimate the standard deviation of the distribution of the value. The mean is considered to be a better estimation of the real value without any influences than only one measured value.
In the case of the transmission time of a message a lower bound is given. The underlying communication hardware transmits messages with fixed speed. Other computer components can only delay transmission. So the assumption that the transmission time is normal distributed could not be made. In conclusion the mean of the sample is not the best estimation of the real value for transmission time. Available parameters of the sample are the minimum value, the maximum value, the mean and the median.
The maximum value of the sample gives an upper bound for the delay of an message. It may be useful if the application has to guarantee that the message is transported in a certain time or to estimate the time useful for an timeout. In our case it is not useful.
The minimum value is more useful. It is a good estimation for the capabilities of the underlying communication hardware. If the sample is great enough one can assume that there was at least one communication made without any delay caused by the other components. But in our case this does not help much. We need information about the general behaviour of the whole system and not only about the communication hardware.
The mean is the value which minimizes the sum of the quadratic distance to all sample values. But this means that sample values which are very far from the mean influence the mean much more than other closer values. So very unlikely but occurring great delays raises the mean a lot.

The median is the value which minimizes the sum of the distances to all other sample values. Therefore it is not so sensitive to great delays, but still covers all available sample values.
So we choose to measure each PRTT value several times. Than we uses the median of the sample as result of this measurement.
Other available benchmarks use the standard deviation to decide when a sufficient number of measurements are made. We do not use this approach. The standard deviation of the standard deviation itself is very high (Compare [Lic88]). This means that the standard deviation of different samples of the same distribution varies very heavily. So we feel that we should not make decisions based on the change of the standard deviation.

3.3.3 Error Distribution

Because influences beyond our control measured value varies. So when using measured values one has to be aware that this value is not correct. This is called the error of the value. The former subsection shows methods to minimize the error. This subsection deals in brief with the effects occurring while using measured values in computations. (For detail informations see [Lic88])
The standard deviation of a measured value is considered as an extent of the error made. The following rules describes how the standard deviation of measured value affect the standard deviation of an computation result (The computation result is called indirect measured value.)
X, Y describes the sample, Z describes the indirect measured value.
$\Delta x, \Delta y, \Delta z$ describes absolute standard deviation.
$\left|\frac{\Delta x}{\bar{x}}\right|, \left|\frac{\Delta y}{\bar{y}}\right|, \left|\frac{\Delta z}{\bar{z}}\right|$ describes relative standard deviation.

$$Z = X + Y \,/\, Z = X - Y \qquad \Delta z = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

$$Z = CXY \,/\, Z = \frac{X}{Y} \qquad \left|\frac{\Delta z}{\bar{z}}\right| = \sqrt{\left|\frac{\Delta x}{\bar{x}}\right|^2 + \left|\frac{\Delta y}{\bar{y}}\right|^2}$$

$$Z = C X^{\alpha} Y^{\beta} \qquad \left|\frac{\Delta z}{\bar{z}}\right| = \sqrt{\left|\alpha \frac{\Delta x}{\bar{x}}\right| + \left|\beta \frac{\Delta y}{\bar{y}}\right|}$$

In Equation 3.2.3.2 and Equation 3.2.3.3 we sum the round trip time of one message sent and n messages sent. We assume that the variation of transport time which a message suffers, is the same when sending one or when sending n messages. So the absolute error we see while sending n messages is far larger then the absolute error we see when sending only one message. (Note the relative error while sending n packages should be smaller than while sending one package.) So the error of the sum is dominated by the larger error, and we do not expect that the error raises significant because of the equitation.
In Equation 3.2.3.1 the error of the half round trip time and the doubled o should be of the same magnitude. Thus we expect a significant increase in the error because of equitation.

3.3.4 Linear Regression

If we draw the values of Equation 3.2.3.1 or Equation 3.2.3.2 in a coordinate system, we would expect that all points would lie on a line. If we could measure without error, this would be true (assuming the LogGP model is valid), but because of errors the points will deviate. So it would be impossible to fit a linear function when we have more than two points. Even if we have only two points it is unlikely that the linear function is valid. Common practice is to use the function for

which the sum of the quadratic distance of all points to the function is minimized. Three types of distances are used:
The real distance; this means the distance between the point and the nearest point of the function.
The x distance; this means the distance between the point and the nearest point of the function with the same y coordinate.
The y distance, this means the distance between the point and the nearest point of the function with the same x coordinate.
The real distance is used when both the x-coordinate and the y-coordinate represent measured values. The x distance is used when only the x-coordinate represents a measured value. And the y distance is used when only the y-coordinate represents a measured value. The x-coordinates represent in our case the size of the packets. Since this is a value that does not varies while measuring we will only use the y-distance.
If we have n points $P_i=\{x_i,y_i\}$ and want to fit the linear function f(x)=y=a*x+b than

$$a=\frac{\sum_{i=1}^{n} x_i y_i - n\frac{\sum_{i=1}^{n} x_i}{n}\frac{\sum_{i=1}^{n} y_i}{n}}{\sum_{i=1}^{n} {x_i}^2 - n\frac{\sum_{i=1}^{n} x_i}{n}}$$

$$b=\frac{\sum_{i=1}^{n} y_i}{n} - a*\frac{\sum_{i=1}^{n} x_i}{n}$$

For details and proof see [Pap99:Chapter IV.5.3]

3.4 Implementation

This subsection deals with implementation details. A characterisation of the System in which the benchmark was implemented is given. Than the details about the implementation of the micro benchmark and the measurement algorithm are given.

3.4.1 Characteristics of the Existing System

The measurement algorithm was implemented within the Open MPI library [GFB04]. The library consists of several frameworks. The BTL (Byte Transfer Layer) is the framework responsible for accessing the communication hardware. Within the BTL several components are implemented. One for each communication hardware type supported. At runtime each component provides at least one module to access the communication hardware. So the BTL provides an unified interface to all available interconnects. Therefore we decided to establish our LogGP benchmark upon the BTL framework. The BTL provides functions for send receive semantic and functions to use RDMA (remote direct memory access). We only use the functions for the send receive semantic. This is done, because we await slightly different results for RDMA. The LogGP model does not cover this. (It is possible to do he measurement again with the RDMA functions. But this would only complicate the measurement.) Sending is done in two steps. First a descriptor has to be obtained, then the sending can be initiated. Receiving is done asynchronous. A callback function has to be registered before the first message could be received. It is called on any incoming message. For data transfer between the callback function and the rest of the program it is possible to provide a buffer. The pointer to this buffer is passed to the register function, and will then be passed to the callback function.

3.4.2 Implementing the PRTT Micro Benchmark

To implement our benchmark it is necessary to change the algorithm. We use the btl_alloc function to obtain an descriptor. It returns an descriptor which holds an buffer of the requested size. Any data which should be sent has to be copied into this buffer. Alternatively we could use btl_prepare_src then we would not have to copy data, because the BTL sends form the buffer we made available. Because we do not use the data we send we do not have to copy any data and calling btl_alloc is sufficient. Now send can be called. (Note: The unknown data in the buffer of the descriptor is sent, so we send messages of the requested size.) We then call the progress function so incoming massages could be handled. The callback function we provide for receiving has to reply the message if the process is the mirror, or has to store the time at which the message was received if the process is the sender. As result we implemented the PRTT algorithm outlined in the following pseudo code (simplified).

```
sender_callback(callback_data, incoming_descriptor){
	callback_data.receive_time = get_time()
	callback_data.waiting = false;
}

sender_function(number_packets, delay, size){
/* register the callback function together with a buffer which is available to the callback
function and the active function*/
	btl_register(measure_callback_function, callback_data)
	/* set a flag that we await a message.*/
	callback_data.waiting=true
	start_time=get_time()
	descriptor=btl_alloc(size)
	btl_send(descriptor,mirror_address)
	for(i=0;i<number_packets-1;i++){
		compute d cycles
		descriptor=btl_alloc(size)
		btl_send(descriptor,mirror_address)
	}
	# progress until the callback function signals that it has received a message
	while(callback_data.waiting){
		ompi_progress()
	}
	return callback_data.receive_time-start_time
}

mirror_callback(callback_data,incoming_descriptor){
	/* increment the number of received packets.*/
	callback_data.packet= callback_data.packet +1
	/* if it was the last packet return the acknowledge*/
	if (callback_data.number_packets == callback_data.packet) then{
		descriptor = alloc(incoming_descriptor.size)
		send(descriptor, meassure_address)
	}
}
```

```
mirror_function(number_packets){
        callback_data.packet=0
        callback_data. number_packets=number_packets
        /* register the callback function together with a buffer which is available to the
        callback function and the active function*/
        btl_register(mirror_callback,callback_data)
        /*progress until all packets are received*/
        while(number_packets > callback_data.number_packets){
                progress()
        }
}
```

3.4.3 Implementation of the LogP Benchmark

For measuring we use the first two nodes available. Hence the nodes with the rank one and two in the world communicator. The PRTTs for 1 packet and a user defined number of packets (default 16) is measured. The sizes ranges from 1 to the maximum size the BTL module supports. We do not measure for all sizes. The size used starts with 1 and is then incremented several times by a user defined value (default 2048 bytes) up to the maximum size the BTL module can handle. This linear increase insures that every range is represented with a similar number of results. If we use a geometrical growth like other benchmarks, smaller packages would be over represented. We now have all PRTT values we need to calculate g,G. We then measure one PRTT with a delay which is equal to the gap for a packet with the half maximum size the BTL module supports and a small size (default 1 byte). The result is used to calculated o and L.

3.5 Results

The benchmark results are presented in this subsection. Through comparing the predicted values with the measured values we prove that our approach is correct. To give an better information much more PRRT values are measured and pictured. Thus it is shown that our chosen d and n values do not influence the results.

3.5.1 Used Systems:

Two different test systems were used, each consisting out of two nodes. System A had a Myrinet/GM connection available. System C had two Gigabit Ethernet and one InfiniBand® connection available. For details see Table 3.5.1.1.

System	A	B
CPU	AMD Athlon® Processor	two Dual Core AMD Opteron® 2200MHz 1024 KB Cache
OS	Linux 2.6.9	Linux 2.6.9
Memory	513936 KB	2006196 KB
Interconnects	MYRICOM Inc. Myrinet 2000 Scalable Cluster Interconnect (rev 03)	InfiniBand: Mellanox Technologies MT25208 InfiniHost® III Ex Ethernet controller: Broadcom Corporation NetXtreme BCM5721 Gigabit Ethernet PCI Express

Table 3.5.1.1: Details about the test systems

3.5.2 The PRTT Values

The PRTT micro benchmark results for PRTT_1_0(size) are shown in Figure 3.5.2.2, 3.5.2.1 and 3.5.2.3 for InfiniBand, Myrinet/GM and TCP across Ethernet. On all platforms the PRTT graph has nearly the expected shape. Only for very small messages a non linearity is present.

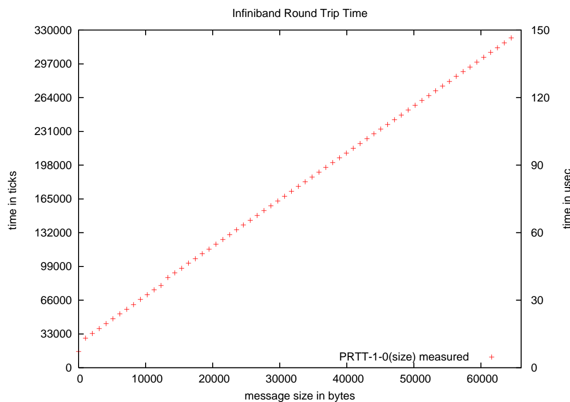


Figure 3.5.2.2: round trip time InfiniBand

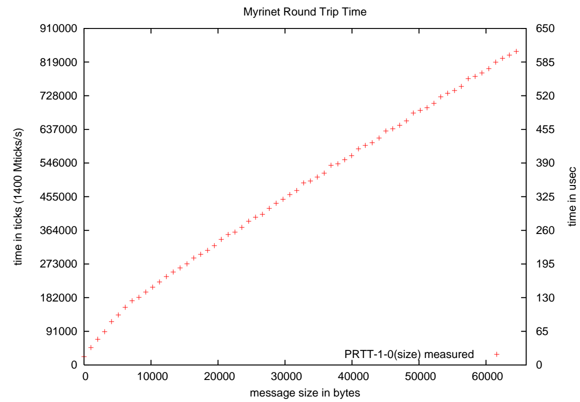


Figure 3.5.2.1: round trip time Myrinet/GM

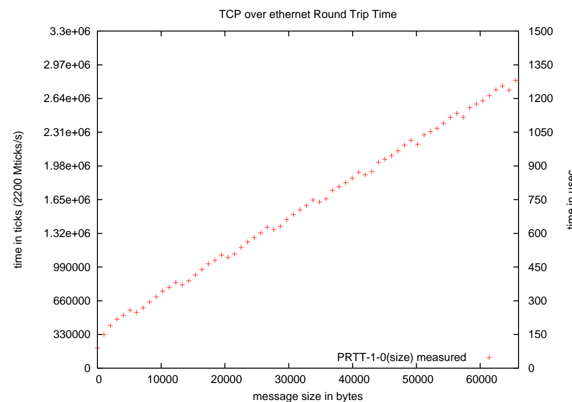


Figure 3.5.2.3: round trip time Ethernet TCP

Figure 3.5.2.4, Figure 3.5.2.5 and Figure 3.5.2.6 shows the PRTT micro benchmark results for a constant number of messages and different delays. As expected, we could see, that for large sized messages the delay does not influence the result. For small sized messages the slope of the PRTT values with delay is , like we expected, similar to the slope of the round trip time for only one message. Unexpected is that we see an increase in the communication time when the sum of delay and overhead differs only lightly from the gap. This means that the communication get influenced in this special case. So while measuring o we have to make sure that d is significantly larger than the gap. However we should use a small d to speed up measurement.

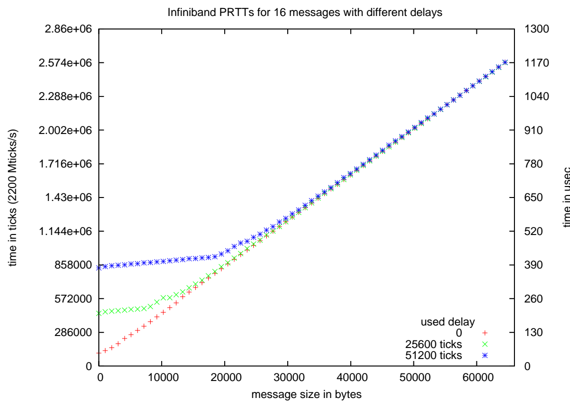


Figure 3.5.2.4: PRTT_16_d InfiniBand

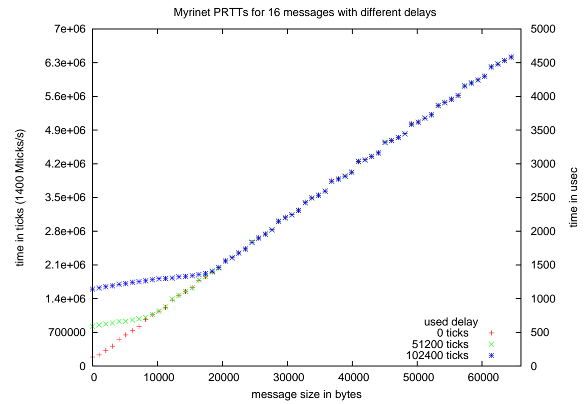


Figure 3.5.2.5: PRTT_16_d Myrinet/GM

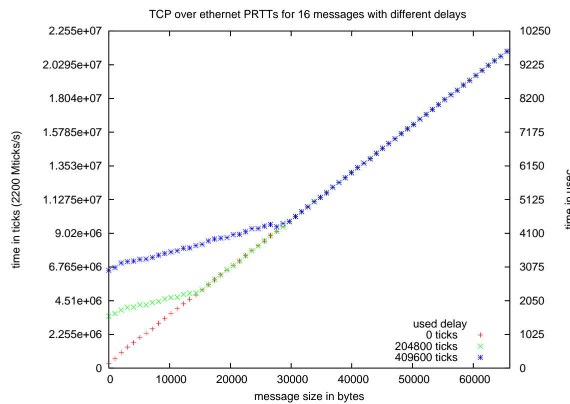


Figure 3.5.2.6: PRTT_16_d Ethernet TCP

3.5.3 The resulting LogGP parameters

Equation 3.2.3.2 was used to calculated the values shown in Figure 3.5.3.1, Figure 3.5.3.2 and Figure 3.5.3.3. A linear function was fitted to assess the g and G values. The G values are in no way influenced by the number of used messages. g is influenced. We could not yet figure out the reason. Only for very small messages g is the dominant parameter. We experienced that for such messages the overhead is more important than the gap. For larger messages G is the dominat parameter. So a difference in g does not significantly influence the scheduling we want to use.

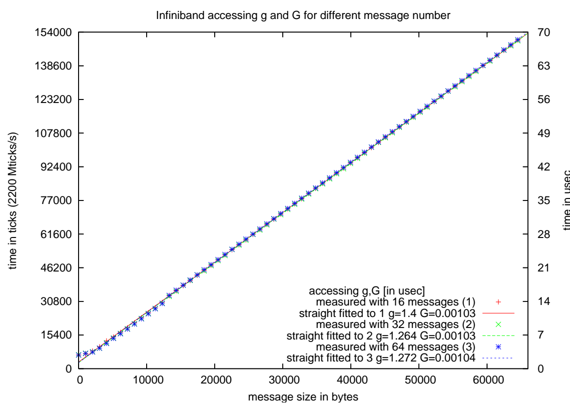


Figure 3.5.3.1: g,G InfiniBand

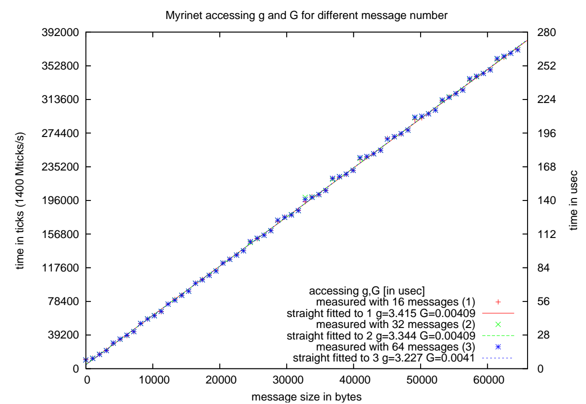


Figure 3.5.3.2: g,G Myrinet/GM

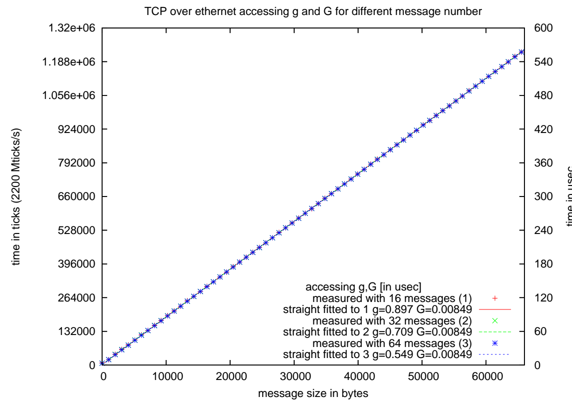


Figure 3.5.3.3: g,G Ethernet TCP

To calculate the o parameter Equation 3.2.3.3 was used. $PRTT_{16_d}$ was the base for the o parameter showed in Figure 3.5.3.4, Figure 3.5.3.5 and Figure 3.5.3.6. As expected the o parameter is not influenced significantly by the used delay. For TCP the overhead increases with the message size. This is because TCP makes an additional memory copy.

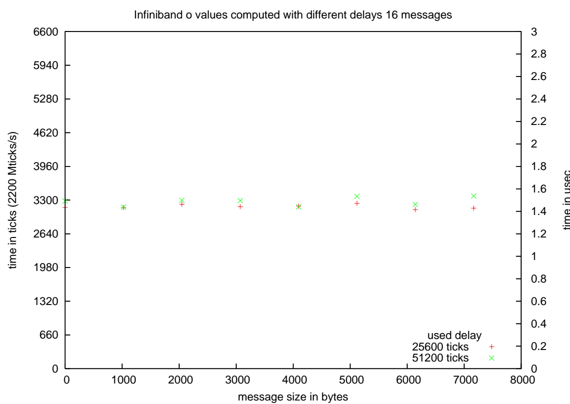


Figure 3.5.3.4: o InfiniBand

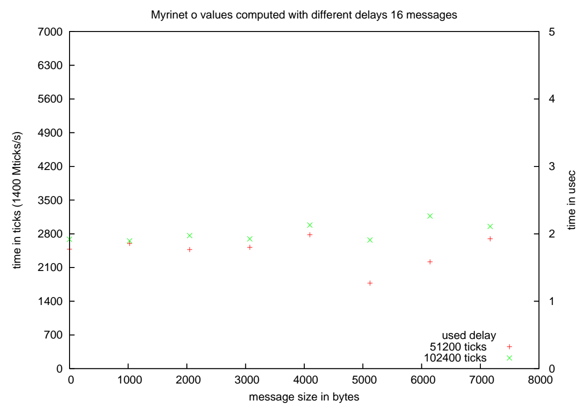


Figure 3.5.3.5: o Myrinet/GM

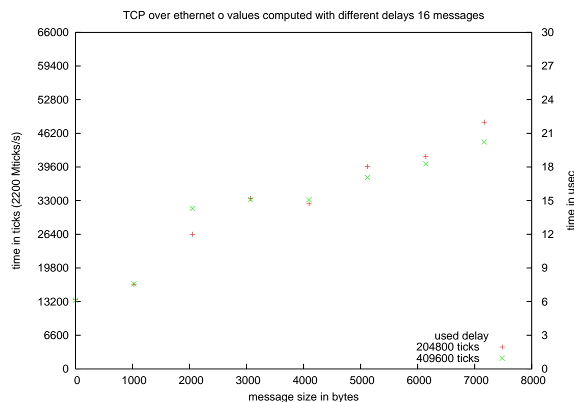


Figure 3.5.3.6: o Ethernet TCP

Table 3.5.3.1 presents the resulting LogGP parameter for InfiniBand, Myrinet/GM and TCP across

Gigabit Ethernet.

	InfniBand	**Myrinet/GM**	**Gigabit Ethernet (TCP)**
g	1,4	3,4	0,897
G	0,00103	0,00409	0,00849
o	1,4	1,9	6,1
L	2,82	24,3	65,4

Table 3.5.3.1: The LogGP parameter (all values are microseconds)

3.6 Conclusion and future work

We compared well known Log(G)P measurements methods and derived a new accurate LogGP parameter measurement scheme. Our method was implemented and evaluated in Open MPI. It measures the LogGP parameters without saturating the communication network in all cases. The PRTT micro benchmark provides the expected results. Only for very small messages and when the overhead is like the gap, do we see unexpected behaviour. Since LogGP is only a model we could not expect that is covers all cases of communication. So we assume that the differences between expected and real behaviour is because of an inaccuracy with the LogGP model. Nevertheless our implementation gives reasonable results for a several communication interconnects. But improvement is still possible. We do not distinguish between the overhead at sender and receiver side. Measuring the receiver overhead could be the aim of future work. Also the reduction of the sample size for each measurement could have a strong impact on the performance of our benchmark.

4. Scheduling

Scheduling is necessary, when several tasks or processes should concurrently use a given resource. Common examples:

- Several processes compete for one or several processing units and are assigned to the processing units by the operating system.
- A set of messages should be sent across one or several interconnects.
- A CPU with several processing units passes instructions (micro instructions) to the processing units, and processes them in parallel.
- In a factory the order in which machines should be used or the time when a product should be crafted is determined.

This shows scheduling could be found in many fields. For this reason an overwhelming number of scheduling algorithms were made available. Many approaches were made to create a classification. Nowadays also an overwhelming number of classifications are available. So we will not present an in depth discussion about scheduling. Instead we will later present 4 basic scheduling algorithms which are commonly used for the given problem. They provide the basics for the final used algorithm

4.1 Taxonomy

The task is something that needs a resource to be processed. For example it could be a message/packet which should be sent across an interconnect. We will assume that tasks are equal. For example equal sized messages or messages split in equal sized packets. After processing the task leaves the system.
The resource is something needed to process a task. This mean when a task is assigned to a resource the resource will be under usage for a certain amount of time. Then a new task could be assigned to the resource.
A process which determines the resource which should process a task when the task arrives at the system is called *scheduling*.
Note: The given examples for a scheduling algorithm uses the terms mostly in a different sense. There is only one resource available. And the task will use the resource for a certain time but then will stay in the system and use the resource later again. So scheduling determines the next process which could use the resource, and but the resource for the process.

4.2 Current Scheduling

For scheduling in Open MPI two different approaches are followed. One for “small” messages and one for “large” messages. An explanation of the different frameworks and the terms component and module follows in subsection 5.2
For small messages functionality of the BML framework is used. The BML provides for each process three arrays of BTL modules (exactly pointer to the modules). One with BTL modules usable for small messages (eager list), one with BTL modules for default communication operations (send list) and one with BTL modules for RDMA operation. The BML framework provides further a get_next functions. It returns one BTL module after the other out of one of the arrays. This function is used to obtain the BTL module for each communication operation for small messages. In a round robin manner (see subsection 4.3.4.1) messages to one processes are sent. This has the advantage of being an very easy and fast scheduling algorithm and the current state of the network is not important for the algorithm.
Although the BML prepares the scheduling, for large messages scheduling takes place in the layers

above the BML. The BML delivers for usable BTL modules a weighting. The weighting is the ratio between the bandwidth of each single BTL module to the sum of the bandwidth available to reach the given process. Large messages are shared between all BTL modules of the sent list, if sent receive semantic is used, or the RDMA list, if RDMA semantic is used in a weighted fair queing manner (see subsection 4.3.4.3). The BTL modules out of the selected list are used one after another. If the remaining message is longer than the eager limit, the BTL module has to process its share of the remaining message. The share is calculated out of the size*weight. Otherwise the BTL has to deliver the remaining message.
For RDMA it is necessary to provide information which memory to use. For this reason a rendezvouz protocol is used. The node which initiates the RDMA request sends a message to the remote node. The remote node does the scheduling and selects the BTL module for each part of the message.

4.3 Evaluation Scheduling Strategies

In this subsection different scheduling strategies are compared with each other. First a definition of optimal scheduling is given then the current implementation is rated and at last some simple scheduling algorithms are presented which will serve as base for the future work.

4.3.1 Optimal Scheduling

In this subsection we only discuss how the schedule to a given set of messages should be optimal. We will not discuss how to reach this schedule.
The aim of the scheduling is to minimize the time a set of messages needs to be transmitted to the receiver. We consider scheduling as optimal, when for a given set of communication operations all interconnects are used in a way such that all interconnects will finish at about the same time. (Finish means that the message has reached the destination.) This would result in the shortest transmission time, because every other possible schedule would mean to reduce the time one or several interconnects needs to complete their communication requests and in turn raise the time other interconnects need.
We only concentrate on interconnects in contrast to other papers like [CFP02],[CFP03]. The communication system is considered to be the bottleneck. The impact of the schedule on other system components like CPU, main memory, and the interconnect for all peripheral components is not considered. So it may be that for a given example an other schedule may be optimal. (In case the interconnects are not the bottleneck.) To address such problems the scheduler has to be aware of the used hardware, and can not be hardware independent. (For example it have to be known how the interconnect is integrated in the system.)

4.3.1.1 Optimal Scheduling for a Single Message.

For one single message an optimal scheduling could be reached through splitting the message into parts, so that every interconnect needs the same time to process its piece.

4.3.1.2 Optimal Scheduling For Several Messages.

The former subsection shows how an optimal schedule would be for a single message. But when processing several messages this approach would be suboptimal. Striping very small messages over several interconnects is very ineffective. Taken the LogGP Model into account we know, that every message causes at least a time g in that the interconnect is not available. So when we send several messages it would be more effective to send each message across one interconnect, thus reducing the number of communication operations and so let the interconnects work more effectively. As a result a scheduling algorithm for small messages would be to decide only how many messages are

sent with each interconnect. It may still be necessary to strip messages. This would be the case when one single large message would utilize an interconnect longer then the other needs to finish their share of messages. In this case the interconnects which would finish earlier have to process parts of the large message.

4.3.2 Real Scheduling

A optimal schedule is commonly impossible, because it is impossible to create any desired packet size. Each packet is a multiple of a basic piece called quantum, commonly the smallest possible quantum is a byte. The share gives the amount of data which should be transmitted under a optimal schedule. The error gives the difference between the amount of data transmitted and the share under real scheduling. This subsection will now give some examples for scheduling algorithms and will rate them according to complexity and the error one has to expect if the algorithm is used.
summarising:

- quantum: the smallest piece a task could be split in
- share: the optimal amount of data which should be processed by one resource
- error: difference between the amount of data currently processed and the share

4.3.3 Evaluation of the Current Scheduling Algorithm

As previously shown in subsection 4.2, Open MPI uses different scheduling strategies for small and large messages. Small messages are never fragmented, whereas large messages are fragmented and the fragments are distributed across the available interconnects in every case. Both seem very reasonable. For an optimal schedule it may be necessary to fragment at least some small messages (For example if only one message should be transmitted). But because of latency the gain would be very small so avoiding fragmentation does not reduce efficiency. Fragmentation of large messages and distributing the fragments across several interconnects could also be suboptimal. For example if two identical interconnects are available, and two messages of the same size should be transmitted, it would be more efficient to send each message across one interface. But current interconnects have to fragment large messages for transmission for their own. So it does not effect efficiency if the scheduling algorithm does this on its own. (In fact it can improve performance when memory registration is taken into account and the message processing is done in a pipeline manner, see [SWB06]).

4.3.3.1 The current algorithm for small messages

The current algorithm for small messages does not take message size into account. This made the scheduling approach suboptimal. It has been widely argued that for small messages the latency is dominating the transmission time so that “small messages” of different sizes can be treated like messages of one size. It is not clear what “small messages” are and when a message is a “large” message. The current Open MPI implementation regards different sizes as thresholds for small messages depending on the used BTL Component.
For example the default setting for the eager limit is for

- openib 12288 bytes,
- gm 32768,
- tcp 65536.

Our measurement results in subsection 3.5.3 show that in every case messages of this size are dominated by the available bandwidth rather then the latency.
Another disadvantage is that the scheduling only take messages into account formally sent to the same target process. This may result in consecutive messages transmitted all across the same interconnect when each message is sent to a different process. For example considering each

process is reachable over the same set of interconnects. Then all eager list will be initialized in the same way. If now one process wants to send one small message to each other process, Open MPI will select for each message the first BTL module and so use only one interconnect. So the scheduling works well only when several messages are sent to the same target process. But this is very unlikely since one large message would serve better.

4.3.3.2 The Current Algorithm for Large Messages

If it is assumed that previous messages were scheduled ideal than all interfaces should be in the same state and will be ready for communication at the same time. Under these circumstances a weighted fair queuing like scheduling for strips of a single message is optimal. It ensures that when all interconnects start transmission at the same time, they will finish at around the same time. Again it is a disadvantage that the scheduler is not aware of transmission to other processes. If a process is reachable over a set of interconnects and another only over a subset, simultaneous transmission to both processes will lead to overuse of the shared subset compared to the other interconnects. For example process A is only reachable over interconnect 1, and process B is reachable over interconnect 1 and 2. We assume that the interconnects are identical. When two large messages of the same size are sent to both processes, the scheduler will send the message to process A across interconnect 1. But it will split the message to process B into two parts sending one across interconnect 1 and the other across interconnect 2. So interconnect 1 has to process three times the amount of data than interconnect 2.

4.3.3.3 Conclusion

It was shown that small messages should not be split. The scheduler should only decide which interconnect is the best choice when a small message has to be transmitted. Fragmenting large messages at any case, does not reduce efficiency, because currently available interconnects will fragment large message on their own. So far the used scheduling is reasonable, but it suffers a lot from the fact that scheduling only takes messages into account which are sent to the same target process like the message under processing. A less but significant problem is that for small messages the message size is ignored.
So a scheduling algorithm should use all available information, both message size and the usage of the interconnects while sending to other targets. Also should the quantum used for scheduling set to value which is far greater than 1 byte but not larger than the maximal fragment size of the used interconnect. In our further discussion we assume that this fragmentation is already done.

4.3.4 Some basic scheduling algorithms

This subsection gives an overview of some simple scheduling strategies, which could be used to develop a efficient scheduling algorithm for small messages.

4.3.4.1 Round Robin

Round robin is the a simple scheduling algorithm. The available resources are put in a list. The first task is assigned to the first resource, the second to the second and so on. When the scheduler reaches the end of the list it starts at the beginning again.

4.3.4.2 Weighted Round Robin

Resources may have different abilities to handle requests. Weighted round robin take this into account. A share / weighting is assigned to every resource. Now tasks are assigned to the resource according to the share. For example three resources A,B,C are available and get the share S1=3, S2=2, S3=1. The first three tasks are assigned to A, the next two to B and the next to C. After this it

starts again with A. Sometimes it is possible to split tasks into several subtasks. Then subtasks are assigned to the resources and so a finer scheduling is possible. This is especially necessary when one task needs a long time to finish and there are only a few other short tasks available.
Weighted round robin could be implemented with a complexity of O(1). This means no matter how many tasks and resources are available the time to assign one task to a resource is constant.

4.3.4.3 Weighted Fair Queuing

The main disadvantage of weighted round robin is that a resource gets all tasks according to their share at once. So it may be that one resource is overused. Weighted fair queuing reduces this error. A “virtual finishing time” is proposed. This time is an estimation when the resource will finish processing the next task. The time the resource needs to finish one task is calculated by dividing 1 by the weighting. So the VFT for every resource is initialized with 1 divided by the share. Every time a task has to be scheduled the resource with the smallest VFT is selected, then 1divided by the share is added to the VFT. Weighted fair queuing could be implemented with a complexity of O(log(N)) (N is the number of available Resources)

4.3.4.4 Virtual Time Round Robin

Virtual Time Round Robin overcomes the runtime problem of weighted fair queuing but provides the same fairness. It is uses different virtual finishing times one for each resource and a global one. For details see [NVZ01] We intended to use it for this work, but realized that it was not possible. The reason is that virtual time round robin only works for a single set of resources, but each process in Open MPI has its own set of BTL modules which could be used to reach it. Maybe [CCN05] gives a better approach. To discuss this could be an aim for future work.

4.3.4.5 Conclusion

Weighted fair queuing and virtual time round robin are scheduling algorithms which achieve a good scheduling with small error and produce only a small overhead. But the share has to be known in advance. Since we want to use the LogGP model, we have the to adapt the algorithm, so that the LogGP parameters are used.

4.4 Developing a Scheduling Algorithm Adapted for Open MPI

At the first stage we will make some assumptions for simplicity. At every stage we will reduce the number of assumptions. We will show some advantages and disadvantage of the proposed final algorithm and will show some further possible improvements, intended for further works.

4.4.1 Simple scheduling

We will assume that:

- every message has the same size.
- only two processes are communicating
- the sender can produce new messages with a infinite rate
- the message transmits in 0 time
- there is only a gap between two consecutive messages
- we use send /receive semantic

There are three possibilities where scheduling can take place:

a. the sender is scheduling
b. the receiver is scheduling
c. both are involved in scheduling

When the receiver is involved in scheduling some information (the final schedule in cases of b.) has to be passed to the sender. This means that before the message could be transmitted, additional packets have to be transmitted. As this work concentrates on small messages all scheduling strategies which need additional transmissions before the message is sent are useless. The time needed to send the additional messages would be enough to send the message itself. Furthermore in cases of additional messages the question of which interface is to use for them has to be solved. So scheduling should only take place at the sender for small messages.

4.4.1.1 The Algorithm

We uses the weighted fair queuing as basis.
I is the number of all available Interfaces, i is the number of the current interface with $0<i\leq I$. All available interfaces are put in an array and a virtual finishing time VFT[i] is assigned to every interface. The VFT should represent the time when the interface is ready for next transmission. So we initialize each VFT[i] with 0. Furthermore we assign the time it needs to get ready between two consecutive messages g[i] to each interconnect. This time corresponds to g in the LogGP model. The array is sorted according to g with the interconnect with the smallest g in first place. Then for each message the interface with the smallest VFT and which is highest in the array will be selected. Finally VFT of the selected interface is adjusted by adding g.

```
/* INIT */
measure g
sort array of interfaces after g
for i=1 to I{
	VFT[i]=0;
}
/*Scheduling */
while(message = get_next_message){
	min= max_int;
	for i=1 to I{
		if min > VFT[i]{
			min = VFT[i]
			selected_Interface=i
		}
	}
	send(selected_Interface)
	VFT[selected_Interface]+=g[selected_Interface]
}
```

4.4.2 Scheduling for Packets of Different Size

In subsection 4.3.3.3 we explained why dealing with all messages as if they have the same size is not reasonable. So we use the G of the LogGP model in addition. So g only gives the time the interconnect is busy between consecutive one byte messages, and G gives the amount of time the interconnect is busy with every byte of the message. So we assign g and G to each interface and adjust the VFT after each transmission with VFT += g+G*(size-1). Now the sorting becomes difficult. For simplicity we assume that the g and G are of same quality, meaning that a interface with small g also have a small G. Later on the sorting will be useless so we will not go into this in detail.

```
/* INIT */
measure g,G of each interface
sort array of interfaces after g
for i=1 to I{
        VFT[i]=0;}
/*Scheduling */
while(message = get_next_message){
        min= max_int;
        for i=1 to I{
                if min > VFT[i]{
                        min=VFT[i]
                        selected_Interface=i
                }
        }
        send(selected_Interface)
        VFT[selected_Interface]+=g[selected_Interface]+G[selected_Interface]*(size-1)
}
```

4.4.3 Scheduling Aware of Pauses Between Consecutive Messages

Obviously no node can produce messages at infinite rate. The o parameter gives the time the processor is busy with processing a message. So it gives a lower bound for the time needed to get the next message ready for transmission. We evaluate some approaches which uses the o parameter to estimated the time the processor is busy between consecutive messages. But these approaches do not take into account that the application using MPI may need some time to provide the messages itself. So we finally decided to use the timer framework Open MPI provides. The VFT as we currently use estimates the time an interface is busy with transmission. So the VFT is only relevant for scheduling when the interface is still busy. So if more time passed than interconnects needed to finish transmissions, the first interface with a VFT smaller than the actual time should be used.

```
/* INIT */
measure g,G of each interface
sort array of interfaces after g
for i=1 to I{
        VFT[i]=0;}
/*Scheduling */
while(message = get_next_message){
        min= max_int;
        time = get_time();
        for i=1 to I{
                if (min > max(VFT[i],time)){
                        min=max(VFT[i],time)
                        selected_Interface=i
                }
        }
        send(selected_Interface)
        VFT[selected_Interface]+=g[selected_Interface]+G[selected_Interface]*(size-1)
}
```

4.4.4 Scheduling Aware of the Latency

The current scheduling algorithm selects the interface which can process the message first. If more than one is available the most suitable is selected according to the g values and so according to bandwidth. But the most suitable interface is not the interface with the best bandwidth but the interfaces which could deliver the message first. The time t_deliver the interconnect needs to deliver one message is, according to the LogGP model, t_deliver=o+L+o+G*size. So we select the Interface which can deliver the message first. Now sorting the array at the beginning is no longer necessary. Interfaces which can deliver at the same time can be treated as equal.

```
/* INIT */
measure g,G,o,L of each interface
for i=1 to I{
        VFT[i]=0;
}
/*Scheduling */
while(message = get_next_message){
        min= max_int;
        time = get_time();
        for i=1 to I{
                if (min > 2*o[i]+L[i]+max(VFT[i],time)){
                        min=2*o[i]+L[i]+max(VFT[i],time)
                        selected_Interface=i
                }
        }
        send(selected_Interface)
        VFT[selected_Interface]+=g[selected_Interface]+G[selected_Interface]*(size-1)
}
```

4.4.5 Scheduling Aware of Parallelism of Overhead and Gap

The aim of the current scheduling algorithm is to select the interface which is likely to deliver the current message first. But the current algorithm may fail, because it takes not into account, that (according to the LogGP model) the overhead may overlap with the gap. In other words the processor may already prepare message for sending although the used interface is not ready. So the time when the message will reach the destination does not calculate to

T_deliver = max(time,VFT)+2o+L

but to

T_deliver=time+2o+L, VFT>time T_deliver=VFT+(o-min(VFT-time,o)+o+L if VFT < time

or T_deliver=max(time,VFT)+(o-min(max(VFT-time,0),o))+o+L

```
/* INIT */
measure g,G,o,L of each interface
for i=1 to I{
        VFT[i]=0;
}
/*Scheduling */
while(message = get_next_message){
        min = max_int;
        time = get_time();
        for i=1 to I{
                if (min > max(time,VFT)+(o-min(max(VFT-time,0),o))+o+L){
                        min=max(time,VFT)+(o-min(max(VFT-time,0),o))+o+L
                        selected_Interface=i
                }
        }
        send(selected_Interface)
        VFT[selected_Interface]+=g[selected_Interface]+G[selected_Interface]*(size-1)
}
```

4.4.6 Scheduling for More than Two Nodes

The last assumption we made for simplicity is that only two processes are communicating. When we have more than two processes the scheduling algorithm could be left unchanged. Only the place where data is held becomes important. The list with all available interfaces has to be held at a central place accessible at any time. At data structures assigned to the destination process only the pointer to the usable interfaces in the central list must be stored. So the scheduler can selects interfaces out of the central list which are usable to reach the desired process, but the VFT at the central data structure has to be adjusted.

4.4.7 Reducing Runtime of the Scheduling Algorithm

The algorithm based on weighted fair queuing, and has a runtime of O(I) (I is the number of available interfaces). Each item in the list of available interfaces has to be processed once. Weighted fair queuing and Virtual Time Round Robin reduces the complexity by sorting the list and either putting the used interface at a new place with a complexity of O(log(I)) or schedule in such a way that the list stays sorted so the scheduler has a complexity of O(1). Because the data structures assigned to the process holds only pointers to the items in the list, we could not replace the items without changing each pointer belonging to this item in each process data structure. So with keeping the list sorted the runtime could not be reduced. Virtual Time Round Robin does not change the positions of the items in the list. But some constraints must be held while scheduling. This is not possible when the scheduler can only use some items of the list. The approach proposed in [CCN05] may solve this problem. But it is complex, and thus not simply adaptable for this problem. As a result we think that a linear complexity according to the available resources is tolerable. We assume that it is very unlikely that systems are used which have a great number of interconnects available (We assume more than 16 is not possible.). So the list will be very short.
The changes made from subsection 4.4.4 to 4.4.5 have a significant impact on the runtime. So we decide to use the algorithm presented in subsection 4.4.4 and accept the error made. The error can not be greater than the maximum overhead. For larger messages this will be outweighed by the time the interface needs to transmit the message. Small messages on the other hand will profit greatly from a faster scheduling algorithm.

In Algorithm in subsection 4.4.4 only the sum 2*o+L is used. So it is reasonable only to store the result at each interface instead of the single parameters.
The results in subsection 3.5.3 shows that we have to use floating point values to store G with a adequate precision. Since floating point operations are very expensive we decided to scale each value (including the time values) through multiplication with a constant value. (In our case we uses 1024. So we do not have to use the floating point multiplication operation but can use the cheaper left shift operation.) This allows us to use integer values.

4.4.8 The Final Algorithm

```
/* INIT */
measure g,G,o,L of each interface
for i=1 to I{
        VFT[i]=0;
        g[i] *= 1024
        G[i] *= 1024
        o[i] *= 1024
        L[i] *= 1024
}
/*Scheduling */
while(message = get_next_message){
        min= max_int;
        time = leftshift(get_time(),10);
        for i=1 to I{
                if (min > 2o_L[i]+max(VFT[i],time)){
                        min=2o_L[i]+max(VFT[i],time)
                        selected_Interface=i
                }
        }
        send(selected_Interface)
        VFT[selected_Interface]+=g[selected_Interface]+G[selected_Interface]*(size-1)
}
```

4.5 Further improvements

[LVP04] proposes to use adaptive algorithm which adjusts the parameter that describe the interconnects. So no measurement at the beginning is necessary and the scheduling can react when the properties of the interconnect change. (For example due to contention.) The proposed adaptive algorithm is based on the ability of InfiniBand to signal the completion of transmissions. Because our scheduler should operate on several different interconnects we could not use this function. We propose to gain the information through sending additional information with every message. The VFT together with the end to end latency is an estimation of the arrival time at the destination node. The sum of the VFT and the end to end latency (VFTL) could be sent together with the message. So the VFTL of a received message should always be larger than the VFTL of the previous message. When the parameter needs adjustment VFTLs of different messages sent over different interconnects should violate this rule. In this case the receiver could send a message to the sender and so signals that an adjustment is necessary. This approach has some advantages. Additional packages are only sent, when the used parameters are wrong. Other approaches send an

acknowledge packet for every received packet and so cause a high additional load across the interconnect. We do not need any time measurement and no tightly coupled clocks.
Never the less we do not follow this approach for several reasons. First the violation can only be detected when several messages are sent in a row over several interconnects to the same destination node. For small messages this is very unlikely (A single large message would serve better.) Even if this is the case (For example when a very large message is stripped into several smaller messages) the differences between the estimated and the real arrival time have to be very large. This is because a message has to arrive before the former message sent arrives. And at the most it seems very difficult to find a method to obtain all LogGP parameters from the information that could be gathered with this approach. This would mean the approach could only be used to adapt to changes in the behaviour of the interconnect. But this adaption would be very slow, because we need several messages to be sent to one receiver and another to be sent back to the sender. But changes in the interconnect behaviour will only be present as long as the contention situation exists. So a slow detection may be useless because the situation has already changed. We decided to use a static scheduling and obtain the parameter for every interconnect at initialization throughout measurement.

4.6 Classification and Measurement of the LogGP Parameter

The g,G parameter are interconnect dependent. They are a representation of the available bandwidth and should be the same between every possible pair of nodes, as long as the same interconnect technology is used. (For example using a Fast Ethernet and Giagbit Ethernet together in one subnet could be fatal. But seems very unlikely.)
The o parameter is highly node dependent. But the ratio between different interconnects should be the same on every node.
The L parameter is highly dependent on the way through the network. So when sending messages to different target nodes a different latency may occur. But in HPC Clusters mostly a central switch infrastructure is used. So the latency should be the same for all nodes. Large clusters may need a cascade of switches. But this will only have a small effect compared to the effects of the overhead and bandwidth even for small messages.
The former discussion shows, that some of the parameters depend on the target of a transmission. In consequence this would mean, that the parameter have to be obtained for all node pairs and have to be stored. If P remote processes are used and I interconnects are available P*I*4 parameters have to be stored. This would lead to a large memory consumption and would slow down scheduling. (It would be unlikely to find the parameters in the cache.)
So we accept the error because of slightly wrong parameters and store the parameters only at each available interconnect. The introduced error would be small since the mostly dominating g,G value is independent of the destination. Small errors could become fatal if scheduling runs for a long time. Then a large number of very small errors sum up to a large error. But fortunately every time the real time is larger then the VFT of a interface the VFT is set to the current time. So the error of previous calculation no longer effects the scheduling. Every time the node has to calculate for a long time the interconnects will finish the transmissions and the VFT will become smaller than the real time. Since the only purpose of clusters is to solve computational difficult problems it is very likely that this situation will occur regularly. So it is sufficient to measure the values for each interconnect only with one node pair and then distributing them among the other nodes.

5. Implementation

This chapter provides an overview of the current Open MPI library, the details of our implementation and the benchmark results.

5.1 Open MPI Overview

Open MPI consists of three main parts, the Open Run-Time Environment: Open RTE, the Open Portability Abstraction Layer: OPAL and Open MPI.
Open RTE provides services for:

- start up,
- providing information to the user,
- storing and providing administrative information,
- identification and allocation of resources,
- mapping of processes to specific nodes,
- launch of processes,
- communication between the processes during start up,
- data conversion when communicating with nodes of a different architecture and
- error management.

Although Open RTE provides crucial services to Open MPI it has little effect to the point-to-point communication itself. So we will not discuss its design in detail. [CWD05] provides detailed information about the Open RTE architecture, while [GSB06] provides an overview of how Open RTE is used during start up of Open MPI in a heterogeneous environment.
OPAL provides helper services for example:

- portably interaction with the operating system (e.g. memory management, input output operations)
- a C based object management system
- a set of container classes (e.g. lists; free lists; hash tables)

The provided service are not important for this work an so will not be discussed further. For details see [BBG06]
Open RTE and Open MPI both are designed to adapt to different system architectures. They both use the Modular Component Architecture MCA. So we will first discuss the MCA before we will discus Open MPI functionality in depth. Than we will present more information about Open MPI.

5.2 MCA

[SL04] provides a overview over the design goals of MCA. The main goal is that the user can change the behaviour of the MPI library easily during start up and that the library can easily be extended without recompiling or relinking. For this purpose Open MPI is divided into several subsystems dedicated to a special task.
These subsystems are defined as MCA *frameworks* which provide a well defined Application Programming Interface (API). Implementations of each framework are called *components*. Each framework can host several components supporting different hardware or implementing different algorithms. At start up the components which should provide the functionality are selected, either automatically (for example the component which supports the given architecture) or by user parameter (the user may select a component which provides a special algorithm).
MCA defines a basic structure (mca_base_component_t) for component information and function pointers to two functions; one for opening and one for closing the component.

Each selected and runnable component provides one or several *modules*. (For example a component supporting a specific network interface may return several modules when it finds several NICs (Network Interface Card) of this type.) Each module provides function pointers and module specific data. The data structure to hold module information and function pointers has to be defined in the framework API. The concept behind MCA is very similar to the object oriented approach. So to get a basic understanding it is helpful to see the framework as an abstract class, the component as a standard class and the module as an object.
Each MCA framework can provide functions as static functions embedded in the header file describing the API of the framework. There it is also possible to define an own component data structure as a child of the basic MCA component definition. So additional information and function pointer which are important for selecting components can be passed from the component to the framework during initialization. The most important part of the API defined by the framework is the definition of the module prototype data structure. Users of the framework access the services provided by the framework through functions defined in the framework API (called framework functions) or with the help of function pointers which can be found at the module prototype (called module functions).
The component itself is also free to create its own definitions of the module or component data, as extensions of the definitions provided by the framework. For example it is possible to store user parameter passed to the component at the component data structure or for each module at the module.
Figure 5.2.1 gives an example of the structure of an MCA Framework "myFramework". It defines an additional function each component has to provide for initialization. As API for the user it provides a function pointer "process_data" and the maximum and minimum data size in the module data structure. Additionally, a second function pointer is provided. This function should return a status.
Also an example component "myComponent" is shown. At the component some user parameters are additionally stored. Private module data including the status is stored at each module data structure. This should illustrate how the getter semantic known from the object oriented approach could be implemented in MCA.(status as private parameter, and get_status as public getter function.)

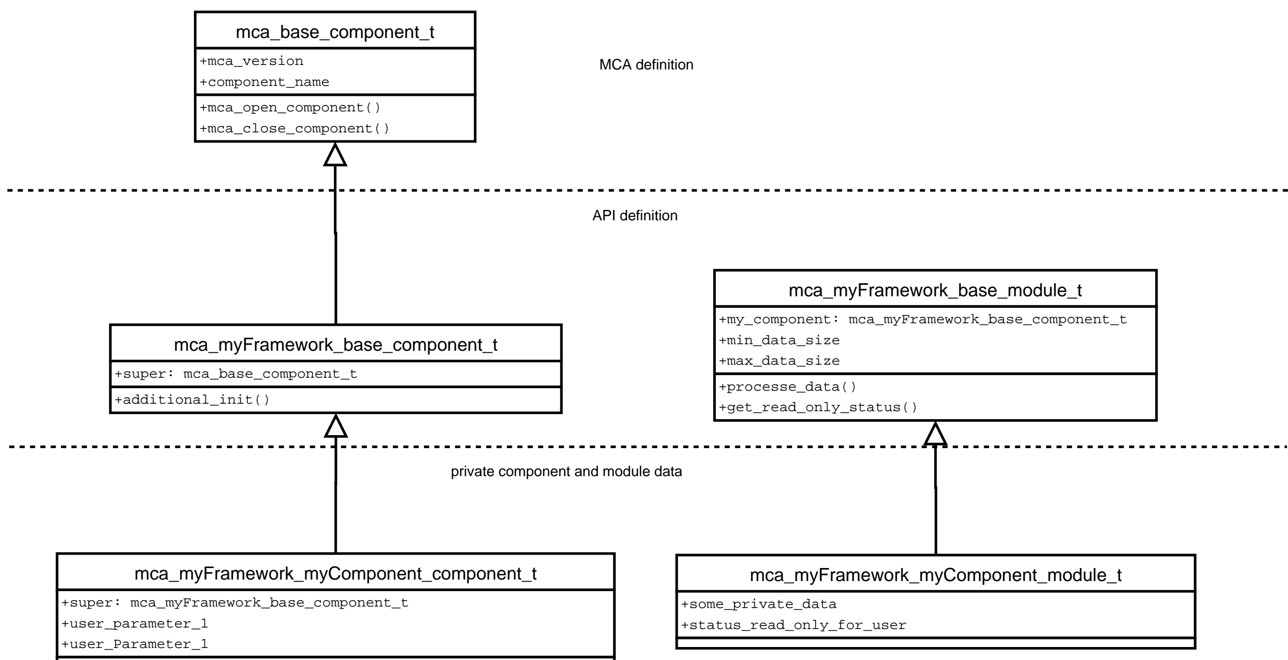
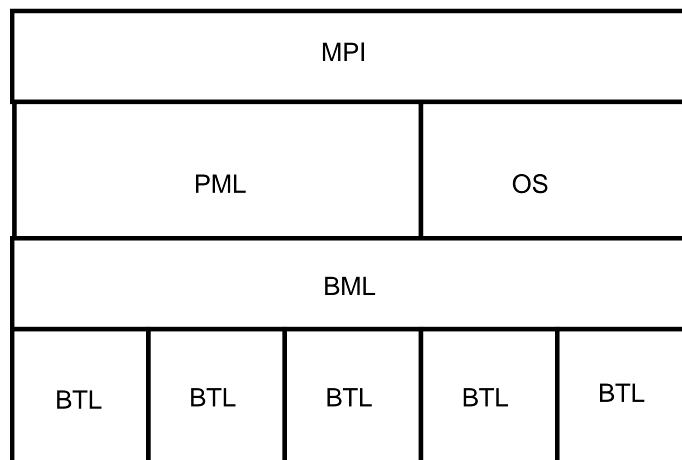


Figure 5.2.1: A MCA framework example

5.3 The Layers of Open MPI Responsible for Communication

The communication architecture consists of five main layers: the Byte Transfer Layer (BTL), the BTL Management Layer (BML), the Point-to-Point Messaging Layer (PML), the One Sided Communication layer (OSC) and the MPI Layer. Two additional frameworks provide support for efficient memory management: the Memory Pool (Mpool) and the Registration Cache (Rcache). All layers are implemented as MCA frameworks except the MPI layer.
The MPI layer provides the MPI functions for the user, and translates it to functionality provided by the MCA frameworks.
The PML and the OSC providing high level communication semantic. These layers are responsible for message scheduling and progression and protocol selection (eager/rendezvous) is also done there. Currently there are two PML components available: the ob1 and the dr component. We concentrate on the ob1 component because it was functional when we started this research.
The BML layer is responsible for discovering and initialization of the BTL modules. Each remote process is associated with a set of BTLs which can reach it. After initializing the PML and OSC use the BTLs directly throughout functions of the BML framework. So no BML module can influence communication.
The BTL framework provides a simple byte mover and so it deems itself to be the basic functionality for communication with both local and remote processes. It provides a simple Interface for several different communication interconnects. Both send/receive semantic and RDMA semantic is supported. The simple interface allows fast development of new BTL components in case new interconnects become available.
The Mpool framework provides memory registration service. It is used from the BTL, PML, OSC and MPI layer. Memory registration is necessary if the RDMA semantic is to be used.
The Rcache framework is used to cache and later search for registered memory. Reuse of registered memory allows a more effective use of the underlying communication hardware.



5.4 Choosing a Framework

Summary of the prerequisites for the scheduling algorithm:

> The aim of the scheduling algorithm is to send each packet across the interconnect which can deliver it first. To achieve this not only static properties of the interconnect but also the current state and usage has to be known. Unfortunately are this information is not directly accessible. The proposed scheduling algorithm estimates the properties with help of knowledge about former transmissions. So the scheduling algorithm needs to be implemented in a way that ensures that it is utilized in each transmission. Also it is necessary to assign the measured and

estimated properties to the available interconnects. Furthermore the algorithm needs the information which process is reachable with which interconnect.

The basic functionality for communication between several Open MPI processes provides the BTL framework. Each BTL module can be regarded as a driver for one kind of communication hardware. So each BTL provides a function to use the hardware the BTL was designed to wok with. Naturally scheduling could not take place here. There are several BTL modules used in parallel depending on the available hardware. Each BTL module is invoked only when the specific hardware should be used, but the scheduling has to be done on every communication operation.
Above the BTL Framework resides the BML Framework. The current implementation helps to manage/create data structures and selects the proper BTL modules to reach each remote process. But the BML module is not directly invoked when communication takes place. So scheduling can not be done here because it has to be done while communicating.
Above the BML layer resides frameworks like the PML framework which are responsible for fragmenting messages and reassembling them, providing a reliable communication and choosing the proper BTL for communicating with a given target process. Because choosing the appropriate BTL is exactly what we want to do while scheduling, the PML framework seems to be the perfect place to implement it. Unfortunately there is no single framework for this functions, but they are implemented in several frameworks (PML, OSC). As stated above scheduling has to take place in a single module which is invoked on every communication request. Because of the design decision to duplicate functionality in several frameworks, we can not implement the scheduling in a framework above the BML without doubling the scheduling functionality. But this would mean that we could not use common data structures easily, which is a prerequisite for the scheduling algorithm.
So we can not find a framework which is suited for scheduling in the way we are looking for.
As a solution we enhance the BML framework's ability to function in a way that allow us to ensure that with every communication request functions of the BML module are invoked. So all functions will be implemented in a new BML module.

5.5 Current Implementation of the BML Framework

This subsection deals with the current implementation of the BML framework and the only available component, the R2 component.
The R2 component provides functions for

- opening and closing the component
- initializing modules

Every module provides functions for

- adding / destroying processes
- adding / deleting BTL modules
- registering a callback function
- finalizing the module

Additionally there are functions provided by the framework itself. These functions are invoked during communication and help manage the BML specific data structures.
A more detailed description of all functions follows in the coming sections. The initializing modules and the adding / destroying process section will describe the data structures specified by the framework and the component. Finally the last section will provide an overview over the interaction of other frameworks with the BML framework.

5.5.1 Opening and Closing the R2 Component

Currently does nothing.

5.5.2 Initializing the R2 Modules

The R2 module is designed as singleton, this means that no more then one module can be created. In this function, some parameters of the module are initialized and the type of the child of the mca_bml_endpoint_t class are set.

5.5.2.1 The mca_bml_endpoint_t Class

Each process is represented in the system by an object of the opal_proc_t class. To enable the pml framework to store process depending information on each opal_proc_t object, each object holds a pointer to the object of the pml_proc_t class. But this pointer is used by the BML framework in the first place. The mca_bml_endpoint_t class is an inheritor of the pml_proc_t class. A object is created while initialization of each process and a pointer is stored in each opal_proc_t object.
This is a violation of the framework architecture, because the BML framework inherits a class intended for use of the PML framework and PML components. A problem is that the PML components still needs to store data for each process. As a solution the PML component uses an own class which is a inheritor of the mca_bml_endpoint_t class. Because instantiation of the objects take place in the BML framework the PML component passes its own class to the BML framework and the BML framework creates an object of the class defined in the PML component.
Figure 5.5.2.1 shows how the classes depend on each other. Classes in grey are only shown for clarity of the over all architecture. All classes are simplified and show only the important parameters. The figure is separated by a dotted line indicating which classes are defined in the PML or BML framework. “myComp” is the name of a example PML component.
Important for scheduling is that each mca_bml_endpoint_t object holds three pointers to mca_bml_base_btl_array_t objects. Each object holds a pointer to the available BTL modules.

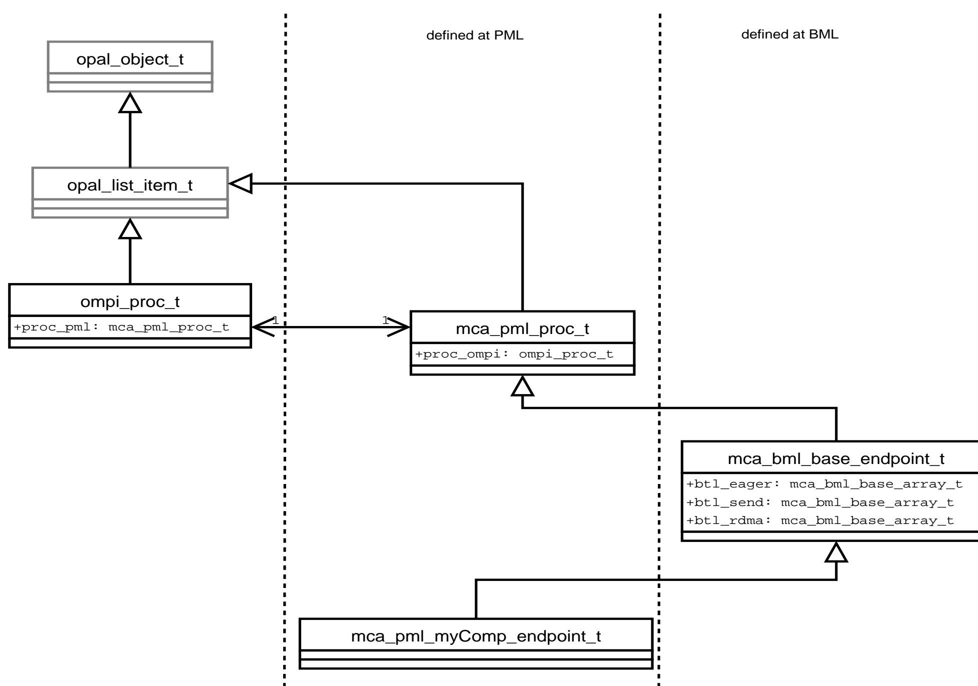


Figure 5.5.2.1: Class dependencies between the PML and BML

5.5.3 Adding / Destroying a Processes

When a new process is created it is announced to the BML layer, by using the function provided by the selected BML module (bml_add_procs, bml_del_procs).
As shown above the process is represented by a ompi_proc_t object. The BML creates an object of the class delivered by the PML component and three mca_bml_base_array_t objects. Each object holds an array of mca_bml_base_btl_t data stuctures. Note the array is not an array of pointers as usual but stores the structures directly.
While constructing the objects the number of available BTL modules is computed and memory is allocated so that every mca_bml_base_array_t object can store one mca_bml_base_btl_t structure for each BTL. Note the methods of the mca_bml_base_array_t class are implemented directly in the bml framework. Then the BTLs suitable for communication with the given process are selected. For each BTL a structure of type mca_bml_base_btl_t is created and inserted into the arrays The structures mainly contain

- a pointer to the BTL module,
- the function pointers provided by the BTL modules,
- copy of the parameters provided by the BTL modules,
- the endpoint provided by the BTL module to address the given process and
- a weighting factor used for scheduling of large messages by the PML.

The addressing information of each BTL module is obtained and stored in the associated structure. Finally the mca_bml_base_endpoint_t structure is attached to the process. An overview is provided by Figure 5.5.3.1.
When a process is destroyed, all objects are destructed and the memory is freed.

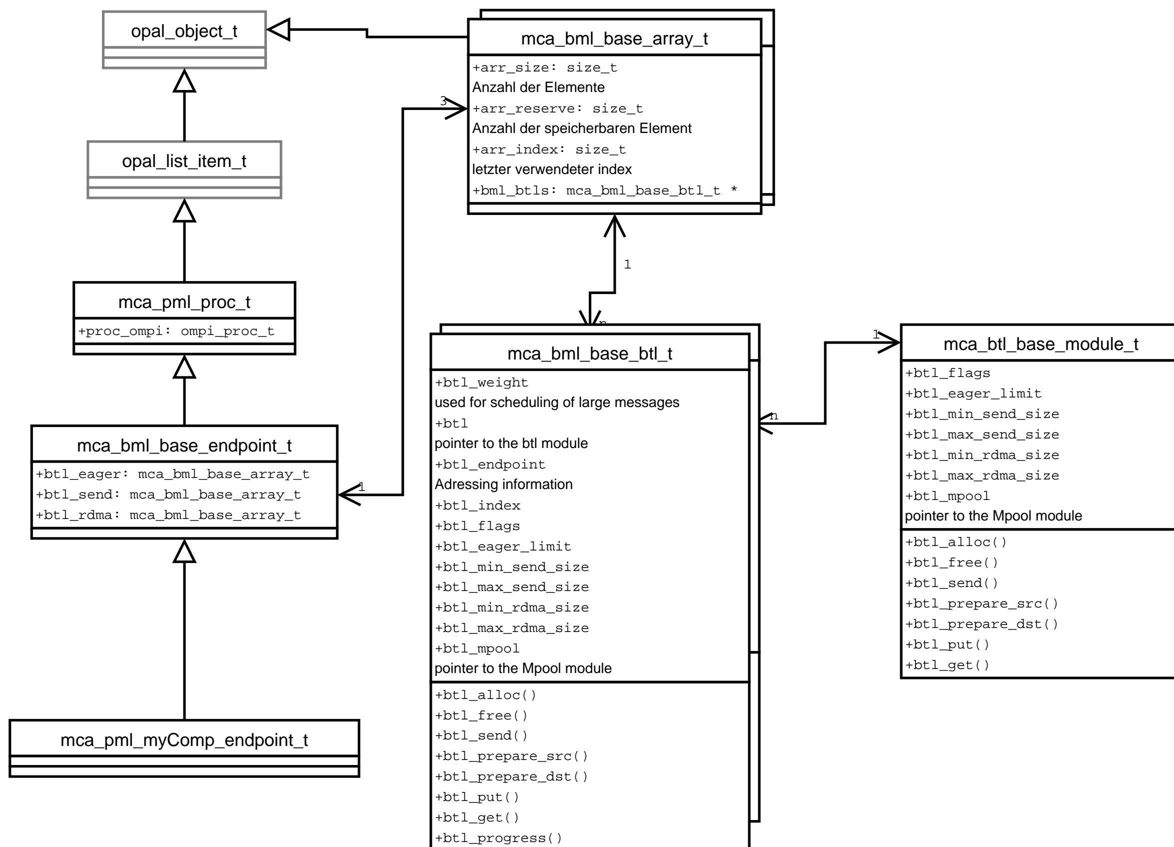


Figure 5.5.3.1: Overview of important BML data structures

5.5.4 Adding / Deleting BTL Modules

Adding BTL modules is currently not implemented.
Is a BTL module no longer available all proc_t objects, each representing a process, are selected. If necessary new structures are created like described in 5.3.1. but this time without the deleted BTL module. They then replace the old structures.

5.5.5 Registering a Callback Function

The BML selects all active BTL modules and uses their functions to register the callback function on each module.

5.5.6 Finalizing the Module

Currently does nothing. Is marked as to do, but there is no hint what should be done in this function. resources allocated by the module should be freed here.

5.5.7 Methods of the mca_bml_base_btl_array_t Class

Objects of the mca_bml_base_btl_array_t class are used in mca_bml_base_endpoint_t objects. To allow other frameworks to access these arrays, the BML framework provides several functions.

5.5.7.1 Reserve

The reserve function takes a number of elements for which an array has to be reserved. It allocates or reallocates memory to store this number of mca_bml_base_btl_t structs. The number of elements for which memory is available is stored in arr_reserve, a private attribute.

5.5.7.2 Insert

Insert returns the pointer to the next free memory in the bml_btls array and increments arr_size, a private attribute.

5.5.7.3 get / set Size

These functions return or set arr_size.

5.5.7.4 get_index / get_next

These functions return the pointer to the element determined by the given index. Get_index takes this index as argument whereas get next uses arr_index a private attribute of the object and increments it. Get_next is used to implement the round robin scheduler above the BML framework. Each task that wishes to send a packet uses get_next.

5.5.7.5 Find

The find function takes a pointer to a BTL module as argument. Then it selects the element in the array which stores the same pointer, and returns the pointer to this element.

5.5.8 Functions Provided by the Framework for Communication

The framework provides functions to simplify access to BTL modules. These functions take a mca_bml_base_btl_t structure as argument. These functions corresponds to the functions provided by the BTL modules and mainly select the endpoint and the BTL module out of the mca_bml_base_endpoint_t structure and call the appropriate BTL function. Available functions are

- mca_bml_base_alloc
- mca_bml_base_prepare_src
- mca_bml_base_prepare_dst
- mca_bml_base_send
- mca_bml_base_get
- mca_bml_base_put
- mca_bml_base_free.

Alloc and prepare are providing descriptors. (For detail information about descriptors see the next section.) Send, put and get uses them for communication. Free finally is used to destruct the descriptor. All functions except alloc and free stores a pointer to the current mca_bml_base_btl_t structure at the descriptor.

5.5.9 Interactions between other Frameworks and the BML while Communicating

Communication is separated in to two steps. First a descriptor has to be obtained which stores the information were to find data for communication. Then the communication operation can be performed.

Modules of the framework above the BML layer select the process for communication and the attached mca_bml_base_endpoint_t structure. Then they select one mca_bml_base_btl_t structure out of the array objects provided by the mca_bml_base_endpoint_t structure. Some modules use the BML framework functions to obtain the descriptors needed for the communication operation and others select the BTL and the function pointer out of the mca_bml_base_btl_t structure to use them. All currently available modules use the framework functions to perform the communication operation. Importantly it should be noticed that each descriptor can only be used with the BTL module that created it.

For diagnostic purposes some modules select the function pointer stored in the BTL module directly and use it. So three means of accessing functions are used.

- the functions provided by the BML framework are used
- the function pointers provided in the mca_bml_base_btl_t structure are used, bypassing the BML framework
- the function pointers provided in the BTL module are used, bypassing the BML as a whole.

Some modules uses different ways for different functions. And different modules use different ways for the same function. So which way is preferred under which circumstances is not visible to us.

5.5.9.1 The Mpool Module

Every BTL provides an Mpool module. This module can be used to access memory management and is used to register memory for RDMA operations. The BML provides a pointer in each mca_bml_base_btl_t struct to the Mpool module corresponding to the BTL. The frameworks above the BML framework use them directly. It is important that RDMA operations are impossible if the Mpool module is not accessible, but each module can only be used with the BTL module it is assigned with.

5.5.9.2 The Receiving Path

In the current implementation receive operations take place without interactions of the BML. Every time a BTL receives a message, the callback functions of the components above the BML framework are called. So the frameworks above the BML are involved directly. If they have to know which mca_bml_base_btl_t struct corresponds to the btl module which initiated the callback has to use the find function described in section 5.5.7.5.

5.6 Implementing Scheduling in the BML Framework

This subsection shows the basic concepts that allow us to implement the scheduling completely into the BML framework. The concept was implemented and tested in a BML component called FSBML.
As shown in subsection 5.4 it is necessary to have information about the current usage of each interface globally accessible within the FSBML module. Also it is necessary that for each communication request functions of the FSBML component are called. The following subsections show how this was achieved and give a deeper description of the additional functions of the FSBML component.

5.6.1 Global Access to the Scheduling Information

To make all data accessible for every FSBML function, we designed the FSBML component as singleton. Only one FSBML module could be created and this module is advertised through the central header file (bml_fsbml.h). So all information stored in this module are accessible for all FSBML functions.
We defined mca_bml_fsbml_parameter_t to store the scheduling information for one BTL module. mca_bml_fsbml_store_t then stores this structure together with the pointer to the corresponding BTL module. At the FSBML module a pointer to a array of mca_bml_fsbml_store_t structures is stored. The array contains one structure for each available BTL module. Figure 5.6.1.1 gives the architecture overview.

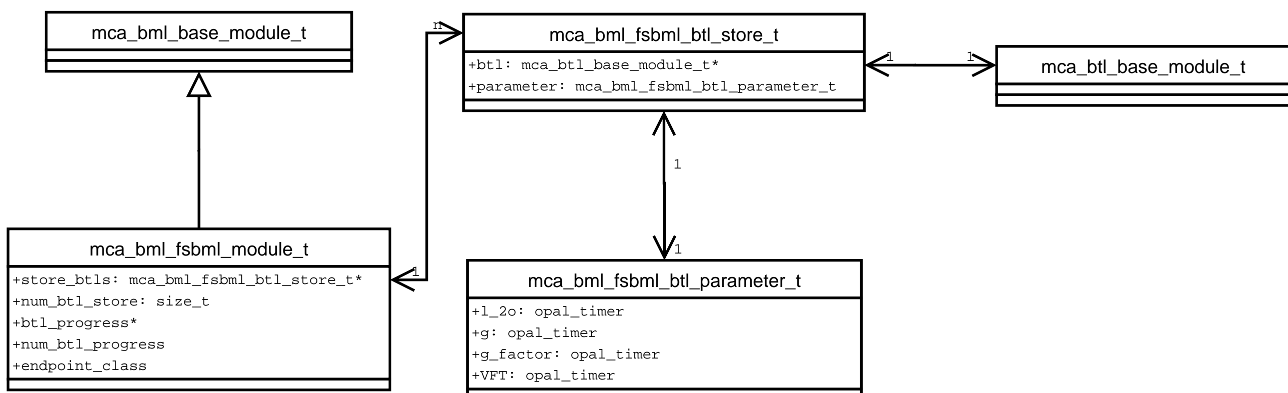


Figure 5.6.1.1: Scheduling information

5.6.2 Involving the BML Component in Communication

The basic idea is to hide the BTL modules (real btl). The FSBML only make our own version of a BTL module (pseudo btl) visible to the above layers. So only function pointers provided by the FSBML are used and the FSBML module is invoked in every communication request.
Since the concept behind accessing functions whilst communicating is not clear we have to support all three possibilities as described in 5.5.9. So we have to create our own

- btl module (pseudo btl) and
- mca_bml_base_btl_t structure (pseudo bml-btl)

for every remote process. And we have to patch the framework functions.
We will only provide one public visible pseudo bml-btl and pseudo btl for every remote process. This ensures that no scheduling is done in the layers above the BML Framework. So for the Frameworks about the BML it looks like each process could only be reached by one BTL module. (our pseudo btl).
As we only provide one pseudo bml-btl and pseudo btl the information which BTL module could be used to reach a process is lost. We have to store this information either at the pseudo bml-btl or at the pseudo btl. Since all functions get the used BTL module as parameter we store this information at the pseudo btl.
We extend the pseudo btl by a pointer to an array of mca_bml_base_btl_t structures. So the design is the same as of the R2. But now the array of available BTL modules is not stored at the bml_base_endpoint_t structure but at the pseudo btl.
The mca_bml_base_btl_t structures stored at the pseudo btl are extended by a pointer to the scheduling information. Each mca_bml_base_btl_t structure holds the addressing information and the pointer to the BTL module. So all the information needed is available. Figure 5.6.2.1 gives the used layout.

Figure 5.6.2.1: fsbml layout at each process; green: the extensions

5.6.2.1 Impact of the Violation of the Framework Architecture

Creating our own BTL module violates the given architecture. This may be fatal if another framework tries to use our BTL module with functions provided by the BTL framework. Fortunately the BTL framework provides no functions.

5.6.2.2 Creating the Flags and Size Information

Each BTL module provides information about their capabilities and minimum and maximum sizes for packets they can transmit. Our pseudo BTL has to set this information in a way that ensures that all BTL modules can handle incoming requests. So we have to set the logic “and” over all flags. Get the maximum of all minimums and the minimum of all maximums.

5.6.2.3 Fragmentation of Messages

For scheduling it would be optimal to use very small packets. This would ensure that all available communication interfaces could work in parallel. Unfortunately current communication interfaces do perform much better when transmitting large sized packets than when transmitting small sized packets. The performance increases with the packet size until it reaches a fixed limit. So a large message should be fragmented in a way that ensures:

1) all interfaces can work in parallel for as long as possible
2) all packets have a size that uses the interfaces most effectively

The naive approach would be to fragment messages within the BML layer. But this would lead to a substantial additional overhead. Packet fragmentation is already done by the layers above the BML so we can use their abilities. We set the maximum packet size of our pseudo BTL to the smallest possible value where transmission is still effective for all BTL modules involved. As a request we will get enough packets to schedule them in parallel but we will not lose performance because of ineffective transmission.

5.6.3 Obtaining Descriptors

When a descriptor is to be obtained, our scheduling algorithm selects one real_bml_btl out of the array stored within the pseudo_btl. The function of this real_bml_btl is called. Since the descriptor can only work with the BTL which has created it, a pointer to the real_bml_btl is stored within the descriptor. All BML framework functions were changed to ensure that no function is changing this pointer. Now each BML framework function sets the pointer in the descriptor only when it is set to NULL. So the changes at the BML framework are transparent for other BML components

5.6.4 Sending Data

When send is called, the real bml btl the descriptor points at is selected and its function is used. This must be done within the functions of our pseudo btl, because the upper frameworks may decide to invoke these functions directly. But also the framework functions use the bml_btl stored in the descriptor.

5.6.5 Receiving Data

When a callback reaches the frameworks above the BML, they may invoke the find function of the arrays within the mca_bml_endpoint_t structs. The mca_base_array_t class was patched so that every object now provides its own find function. If the find function of the framework could not find the desired structure, it calls the find function of the object. The default function simply returns NULL indicating that no matching structure was found. So the patch is transparent for the other components

The FSBML sets the pointer of each object to its own function. The find function of the FSBML now also searches in the pseudo btl for a matching structure.

5.6.6 Using RDMA

The main goal of this work is to determine if scheduling of messages of small sizes may be done efficiently. Using RDMA creates a large overhead, so common MPI implementations do not use RDMA to send small messages directly out of the user buffer. Despite this it is necessary for RDMA operations to pass the Mpool modules of the BTL moduls to the frameworks above the BML. This could not be done because we only have one pseudo_bml_btl and may only pass one Mpool module. Every BTL component uses their own Mpool component. Furthermore a Mpool module

could only be used with the BTL module that provides it. Creating our own Mpool module which can be passed to the above the BML lying frameworks and can map the functions to the appropriate Mpool modules of the BTL modules which is very difficult and out of the scope of this work.

5.7 Performance Analysis of our Design

The performance of our algorithm is significant for the overall performance and was analysed in subsection 4.4.7. But the concept of data layout and function design may also have an significant impact on performance. In this section we will analyse how much additional function pointers have to be used whilst communicating creating additional overhead. Then we will discuss how many memory segments will be accessed. Every increase makes it less likely to find the data in the CPU cache and may trigger more expensive memory accesses. A memory segment mean here a distinct ranges of memory. We consider memory as distinct when it was obtained with distinct calls to memory allocation functions.

5.7.1 Use of Function Pointers

Every time a function from our pseudo btl is accessed we have an additional use of a function pointer compared with the current implementation. It is necessary when a descriptor has to be obtained because here the scheduling algorithm has to be invoked. All other operations can be done without invoking the functions provided by our pseudo btl. To achieve this the functions provided by the BML framework have to used. They select the real_bml_btl out of the descriptor. So their functions could be used and so the functions of the BTL module are used directly. This means we need per communication operation one additional function call with the help of function pointers when the above lying framework uses the BML framework functions. Or every function call means an additional use of function pointers if the above lying framework uses our pseudo btl directly.

5.7.2 Memory Segments

When the current BML component is used, three memory segments are accessed while communicating. The mca_bml_base_endpoint_t object, the mca_bml_base_array_t object, the array containing all mca_bml_base_btl_t structures, and the btl itself is accessed.
When our component is used, the mca_bml_base_endpoint_t structures, the mca_bml_base_array_t object, the array containing the pseudo bml_btl, the pseudo btl, the real bml btl and the btl itself is accessed. It seems useful to allocated memory for the pseudo bml btl and pseudo btl as a contiguous memory segment. So we use only one memory segment more then the current BML component.

5.7.2.1 Patching the mca_bml_base_array_t Class

The memory for the pseudo bml btl is allocated by the reserve function of the mca_bml_base_array_t class. This function only allocates memory for mca_bml_base_btl_t structures. So if we want to allocate the memory of both the pseudo bml btl and the pseudo btl together, we have to either override the reserve function or patch the mca_bml_array_btl_t class in a way which allows it to deal with elements of different size.
We chose the last one because this patch allows us to deal with the issue without violating the framework architecture. We extended the class by a parameter which holds the size of one single element. Then all functions were changed so they now use this size to calculate the array size and the addresses of each element. When constructing an object the size of the mca_bml_base_btl_t structure is used to initialize the objects size attribute. An additional function allows us to change the value of this parameter as long as the reserve function is not called before. So the changes are transparent for current components, and create a way to tell the array the size of one element.

5.8 Results

In this subsection we will present the results of the benchmarking of our software. First we will present communication performance when communicating across one single network. This is done to prove that the new BML component does not significantly raise the overhead and achieves nearly the same performance like the BML component which does no scheduling.
Next we will present the results when using several interconnects at once. We will show that we achieve the same performance when the available interconnects are equal and perform better, when different interconnects are available.

5.8.1 The Microbenchmark

For testing we used the extensible open source Netgauge tool [Net06]. Netgauge is a modular networking benchmarking tool that uses high-precision timers to benchmark network times. The difference to other tools like NetPipe [38], coNCePTuaL [35] or the Pallas Micro Benchmarks (PMB) [36] is that the the framework offers the possibility to use MPI as an infrastructure to distribute needed protocol or connection information for other low-level APIs (e.g. Sockets, InfniBandTM, SCI, Myrinet/GM ...) or to benchmark MPI Send/MPI Recv itself. We used Netgauge only together with MPI. The LogGP benchmark was used to determine overall performance of the single communication infrastructures and an extended one to one "communication pattern" was used to examine the scheduling performance of our BML component. When Netgauge is run with our "communication patter" it sends a user defined number of messages to the receiver and the receiver sends all messages back. The time for this operation is divided by the number of send packets and so a comparable per packet time and bandwidth is available. The benchmark algorithm:

```
sender:
get start_time.
for(counter=0;counter < N; counter ++){
        send message;
}
for(counter=0;counter < N; counter ++){
        receive message;
}
get end_time
return (end_time – start_time)/N
server:
for(counter=0;counter < N; counter ++){
        receive message;
}
for(counter=0;counter < N; counter ++){
        send message;
}
```

Because this work concentrates on scheduling of small messages we show only the performance of small messages. Our BML component can not use the RDMA abilities of the used Hardware. For this reason it always performed worse than the available R2 component when large messages are used. We will later show how this disadvantage could be overcome.

5.8.2 The Used Test Systems

Three different test systems were available, each consisting out of two nodes. System A had a Gigabit Ethernet connection and a Myrinet/GM connection available. System B had two Gigabit Ethernet connection and two InfiniBand connection available. System C had two Gigabit and one InfiniBand connection available. For details see Table 5.8.2.1.

System	A	B	C
CPU	AMD Athlon Processor	two AMD Opteron 244 1800 Mhz 1024 KB Cache	two Dual Core AMD Opteron 2200MHz 1024 KB Cache
OS	Linux 2.6.9	Linux 2.6.9	Linux 2.6.9
Memory	513936 KB	2056160 KB	2006196 KB
Innterconnects	MYRICOM Inc. Myrinet 2000 Scalable Cluster Interconnect (rev 03) Ethernet controller: SysKonnect SK-98xx V2.0 Gigabit Ethernet Adapter (rev 15)	InfiniBand: Mellanox Technologies MT25208 InfiniHost III Ex Ethernet controller: Broadcom Corporation NetXtreme BCM5721 Gigabit Ethernet PCI Express	InfiniBand: Mellanox Technologies MT25208 InfiniHost III Ex Ethernet controller: Broadcom Corporation NetXtreme BCM5721 Gigabit Ethernet PCI Express

Table 5.8.2.1: Details about the test systems

5.8.3 The Performance with one Network

We show the performance of Open MPI when only one interconnect is available to communicate and no scheduling takes place. We will show the performance of our FSBML component and the R2 component for several numbers of messages (1,8,16,32).
Figure 5.8.3.1 shows the bandwidth measured with Netgauge for different packet numbers on System C. Both the performance of the FSBML and the R2 component are shown. As interconnect one InfiniBand NIC was used. As expected the performance does not change significant when using different BML components.

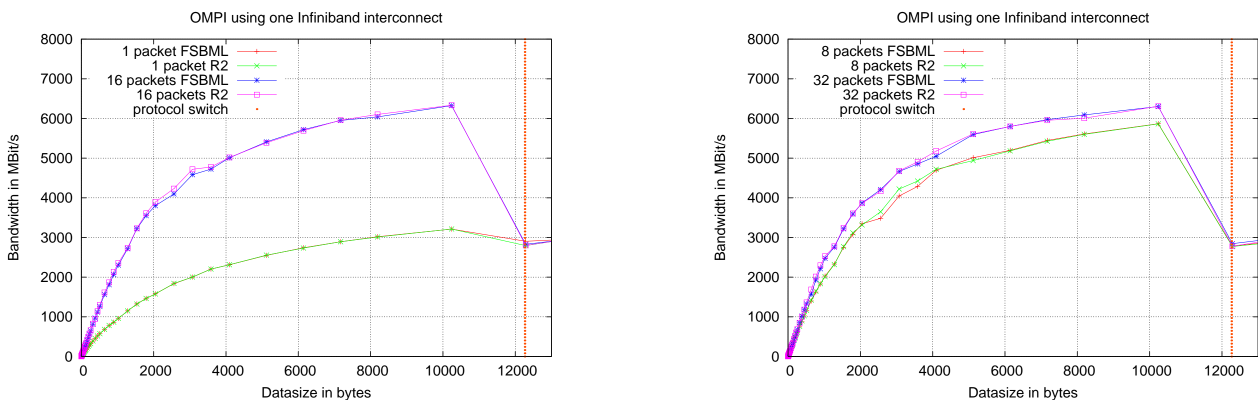


Figure 5.8.3.1: Open MPI using one InfiniBand interconnect

Figure 5.8.3.2 shows the performance when one Myrinet/GM interconnect is available. The bandwidth when using the FSBML and R2 component is again equal.

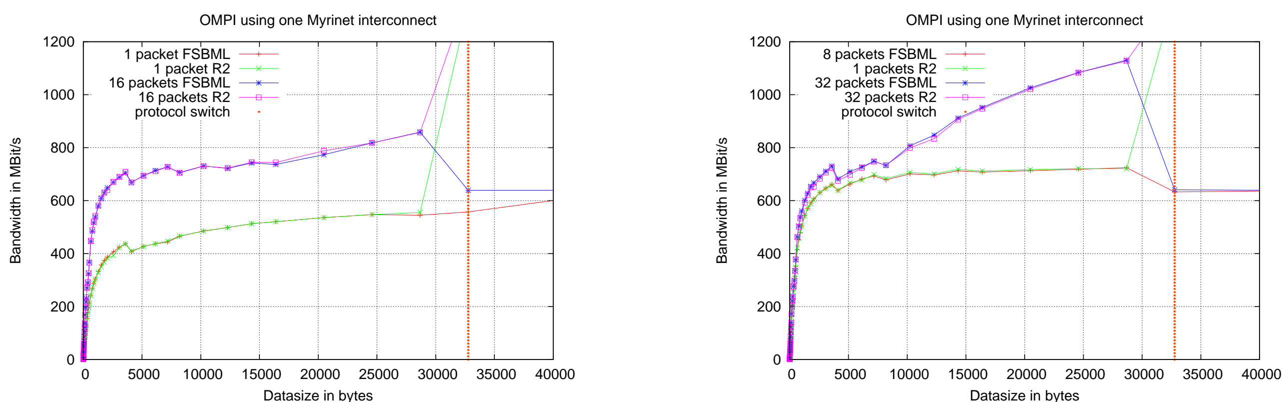


Figure 5.8.3.2: Open MPI using one Myrinet/GM interconnect

In Figure 5.8.3.3 the bandwidth when using TCP with one Gigabit Ethernet is shown. For the same number of packets the performance of FSBML is not significantly distinct from the performance of the R2 component.

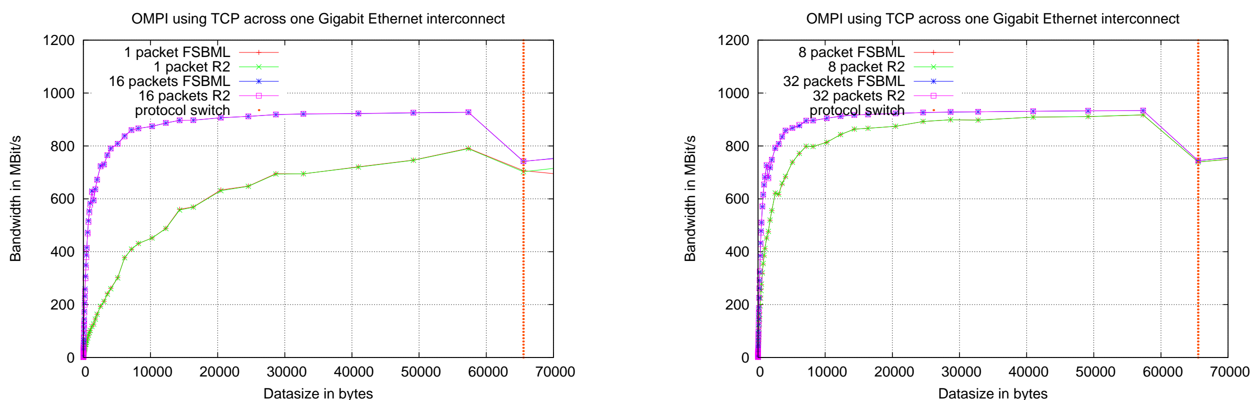


Figure 5.8.3.3: Open MPI using TCP with one available Gigabit Ethernet interconnect

5.8.4 The Performance Using two Identical Networks

For the next test series we used two identical interconnects. Netgauge sends again a number of messages from one process to another. We expect that the scheduling of the PML component outperforms the scheduling of our FSBML component. The simple round robin approach is optimal when sending same size messages to the same process over equal interconnects. The scheduling algorithm implemented in the FSBML should also achieve also an optimal schedule, but because of a low overhead of the scheduling algorithm, the PML should perform better.
Figure 5.8.4.1 shows the performance when two InfiniBand interconnects of system B are used. The expected higher performance when using the R2 component is not visible. In Fact Open MPI perform equally for both components.

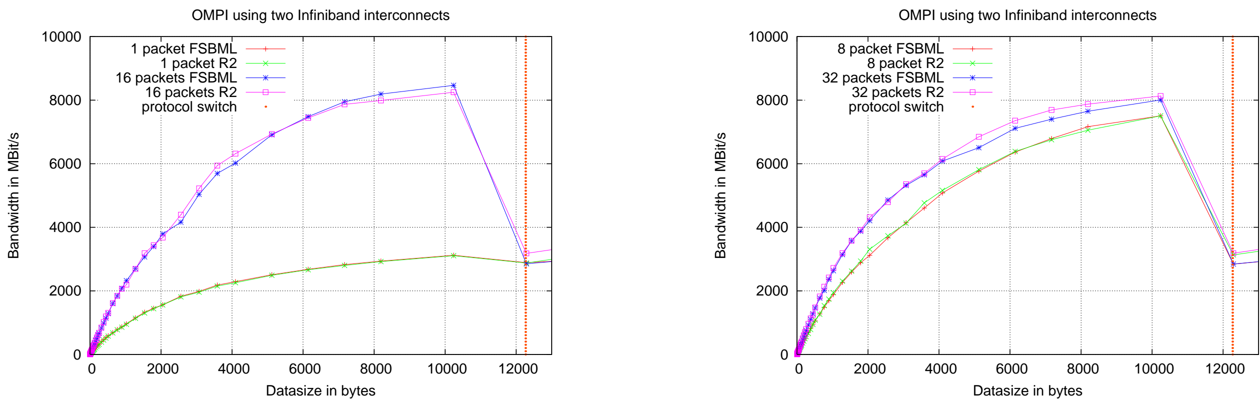


Figure 5.8.4.1: Open MPI performance using two InfiniBand interconnects

In Figure 5.8.4.2 the performance of Open MPI with two available Gigabit Ethernet subnets is shown (tested on ystem C). It performs equally no matter which component is used.

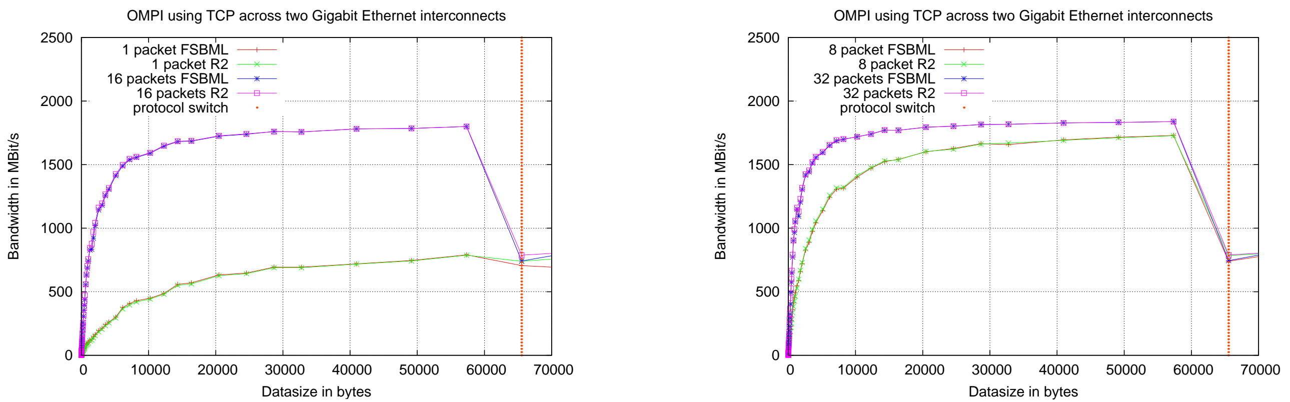
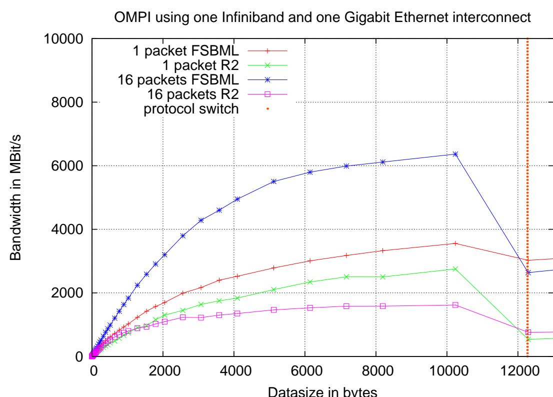


Figure 5.8.4.2: Open MPI performance using TCP across two Gigabit Ethernet interconnects

5.8.5 The performance using two Different Networks

In this subsection we present the benchmark results when using two different interconnects. When using the R2 component the scheduling of the PML is still working. But because the PML only uses a round robin algorithm we expect a very bad performance. Our component prevents any scheduling in the PML component. So we expect a much better performance.
Figure 5.8.5.1 shows the results when an InfiniBand interconnect is used together with a TCP connection across Gigabit Ethernet (system C). As expected the performance when using the R2 component is significantly worse than when using the FSBML component.



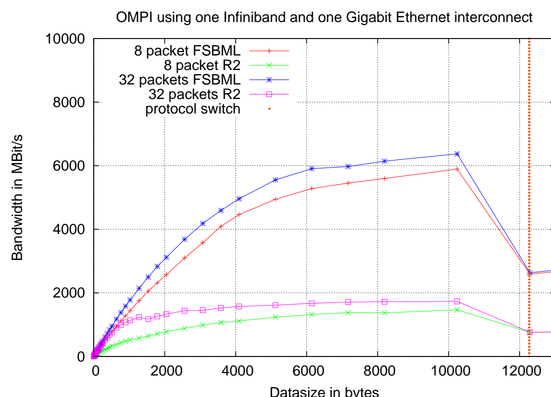
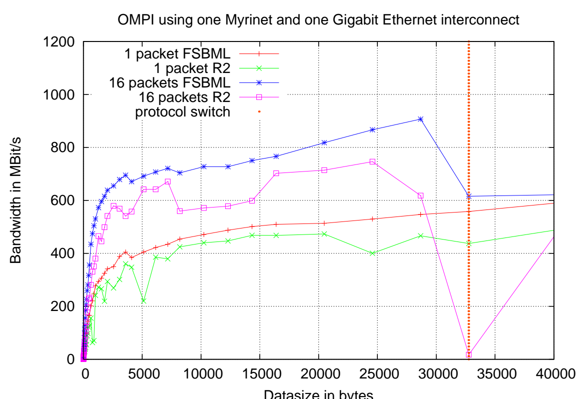


Figure 5.8.5.1: Open MPI performance using one InfiniBand and one TCP Gigabit Ethernet interconnect

In Figure 5.8.5.2 the performance when using Myrinet/GM together with TCP across a broken Gigabit Ethernet connection is used (the TCP connection provides only a bandwidth of 400 MBits). Again the Performance when using the R2 Component is much worse.



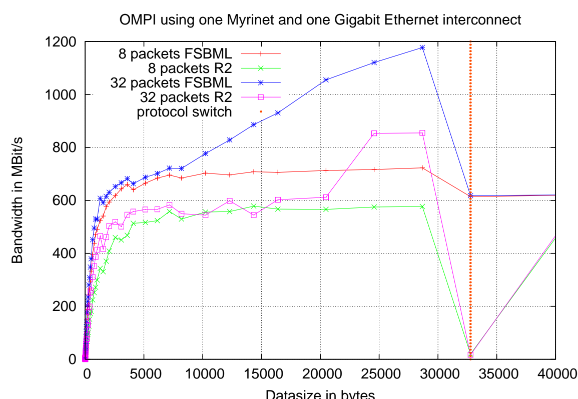
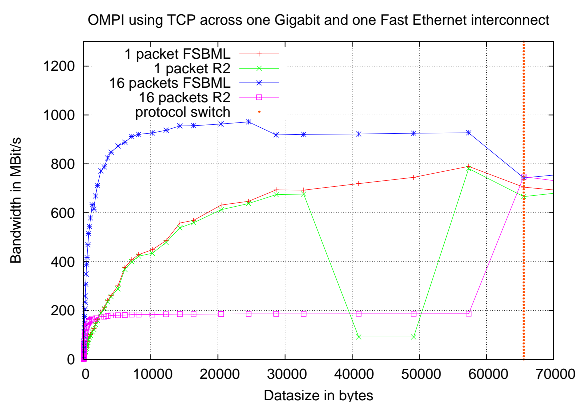


Figure 5.8.5.2: Open MPI performance using one Myrinet/GM and one broken TCP Gigabit interconnect

The performance of a TCP connection across Gigabit Ethernet together with a TCP connection across Fast Ethernet is shown in Figure 5.8.5.3. The performance when using the FSBML component is again much better than the performance when using the R2 component.



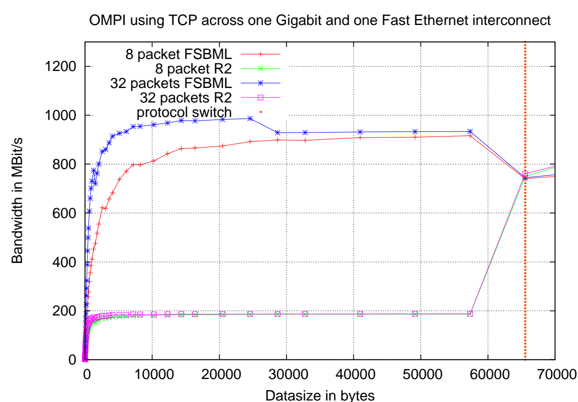


Figure 5.8.5.3: Open MPI performance using TCP across one Gigabit Ethernet interconnect and TCP across one Fast Ethernet interconnect

5.8.6 The Benefit of Scheduling

The former subsection shows that the FSBML component performs much better then the current implementation. But it does not show if the FSBML component uses the available network effectively. This subsection presents a comparison between the performance of Open MPI when one interconnect is available and the performance of Open MPI when two interconnects are used. The FSBML component was used together with two interconnects and the R2 component together with one interconnect.

Figure 5.8.6.1 presents the results when using two InfiniBand interconnects together. At first sight the outcome is disappointing. When using two identically interconnects one could expect that the cumulative bandwidth is nearly double the height of the bandwidth of one interconnect. At the end of this subsection it is shown that in the case of two InfiniBand interconnects the CPU or the memory becomes the bottleneck for small messages(small messages are copied before transmission when the eager protocol is used). For this reason no higher bandwidth than those presented are possible.

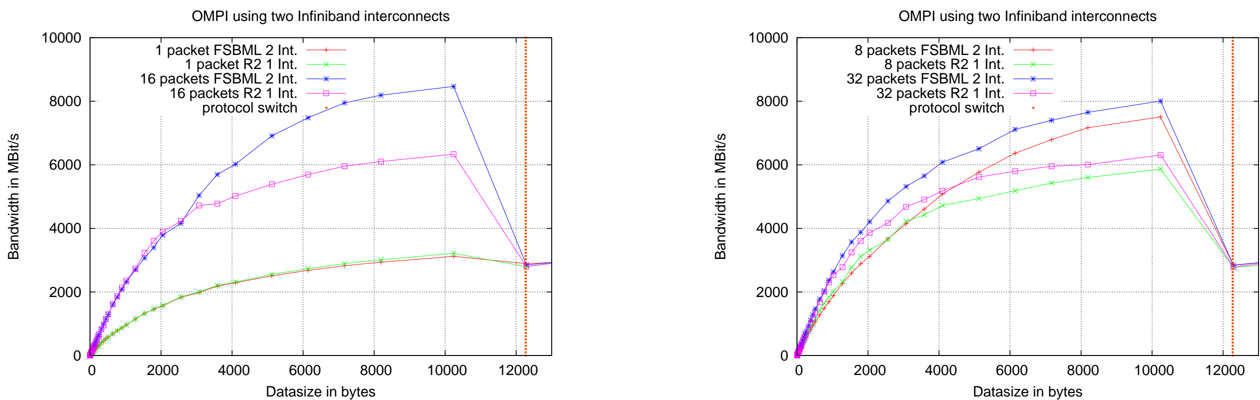


Figure 5.8.6.1: Open MPI using two InfiniBand interconnects compared to using one

In Figure 5.8.6.2 the performance of TCP across two Gigabit Ethernet interconnects is show. The bandwidth nearly doubles when the FSBML component is used.

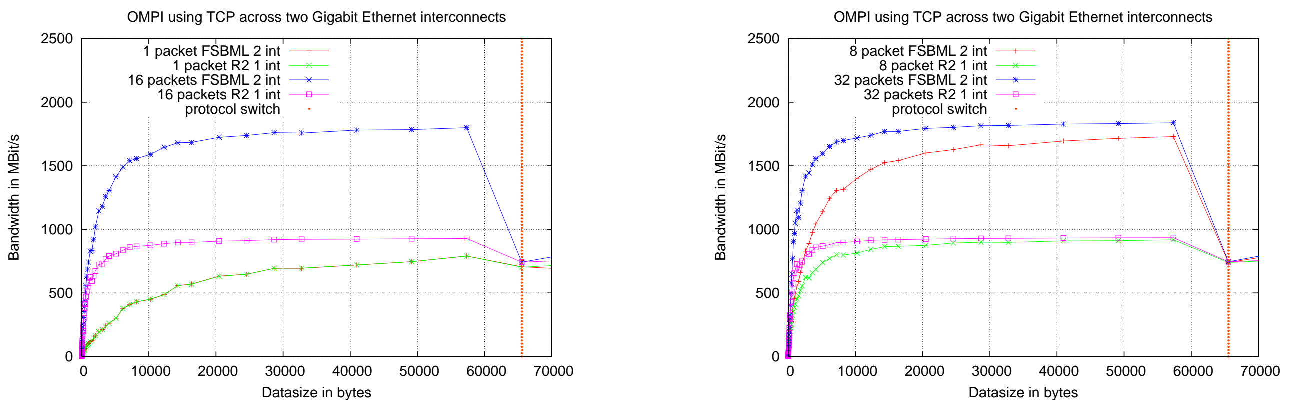


Figure 5.8.6.2: OMPI using two TCP Gigabit Ethernet connections compared to using one

The results of the benchmark with a InfiniBand interconnect used together with a TCP Gigabit Ethernet connection are presented in Figure 5.8.6.3. The bandwidth is not increasing compared to the bandwidth of a single InfiniBand connection. The reason is again the overhead of the interconnections. At the end of this subsection we will show that the overhead produced by TCP is larger than the time the Infiniband interconnect processes a message. So scheduling can not gain an increase in performance.

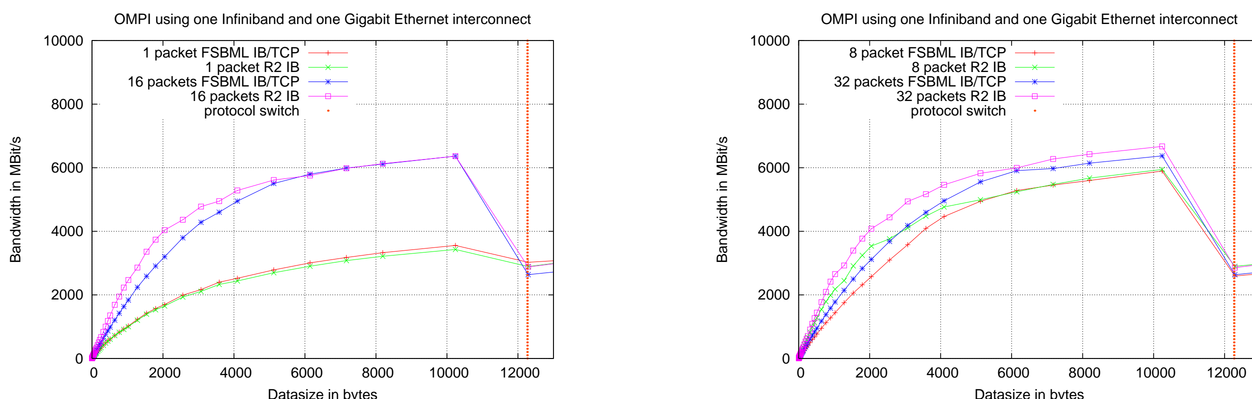


Figure 5.8.6.3: Open MPI using InfiniBand together with TCP across Gigabit Ethernet compared to using one single InfiniBand interconnect

Unfortunately the Gigabit Ethernet network used together with Myrinet/GM was broken. It provides only a small bandwidth of around 450 MBits, whereas the Myrinet/GM interconnect can provide a bandwidth of around 2 GBits. Furthermore is the performance of the TCP connection very inconstant. Thus scheduling is very difficult and can only provide a small increase in performance. The results presented in Figure 5.8.6.4 show subsequently a significant performance increase.

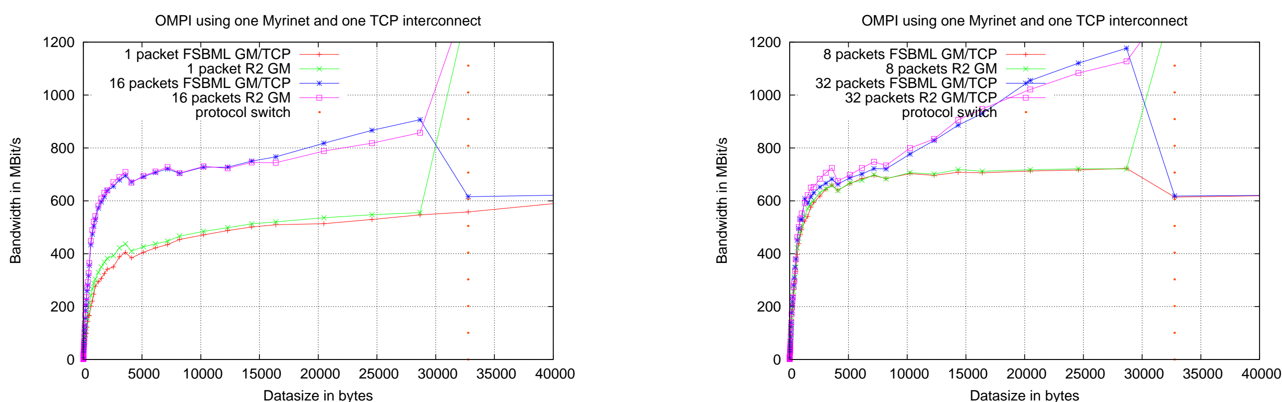


Figure 5.8.6.4: Open MPI using Myrinet/GM together with TCP across Gigabit Ethernet compared to using one single Myrinet/GM interconnect

When using Gigabit Ethernet together with Fast Ethernet one has to be aware that the capabilities of them differ very strongly. The bandwidth Fast Ethernet provides is only one tenth of the bandwidth which Gigabit Ethernet provides. So the best theoretical possible performance gain is about 10 percent. But this would only be possible for very large messages. For small messages the latency and overhead are also important, so in practice it is very difficult to obtain any performance increase, which could countervail the loss of performance because of scheduling. Nevertheless when using the FSBML component the benchmark results presented in Figure 5.8.6.5 shows a small but visible performance increase.

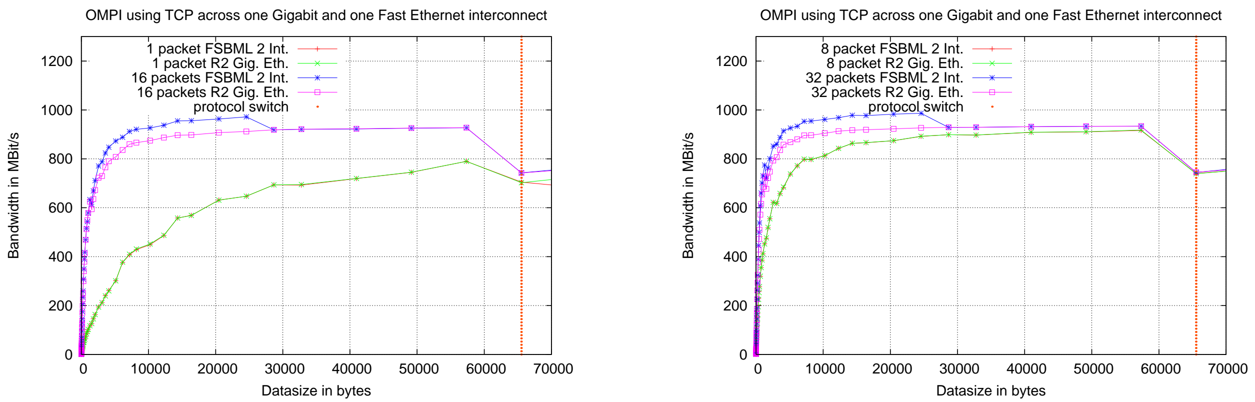


Figure 5.8.6.5: Open MPI using TCP across one Gigabit Ethernet interconnect together with TCP across one Fast Ethernet interconect compared to using one single Gigabit Ethernet interconnect

Netgauge also provides a module which can be used to obtain LogGP parameters. As Netgauge may be used together with a MPI library we used it to obtain the parameter when using Open MPI. The resulting o and g,G parameters are presented in Figure 5.8.6.6 for both a InfiniBand interconnect and a TCP connection using Gigabit Ethernet. The sum: g+G*(size-1) can be interpreted as the time the interconnect needs to process a message. When several interconnects are used together this time will reduce, because the interconnects can work in parallel. The o can be interpreted as the time the library needs to process the message. This time can not be reduced.

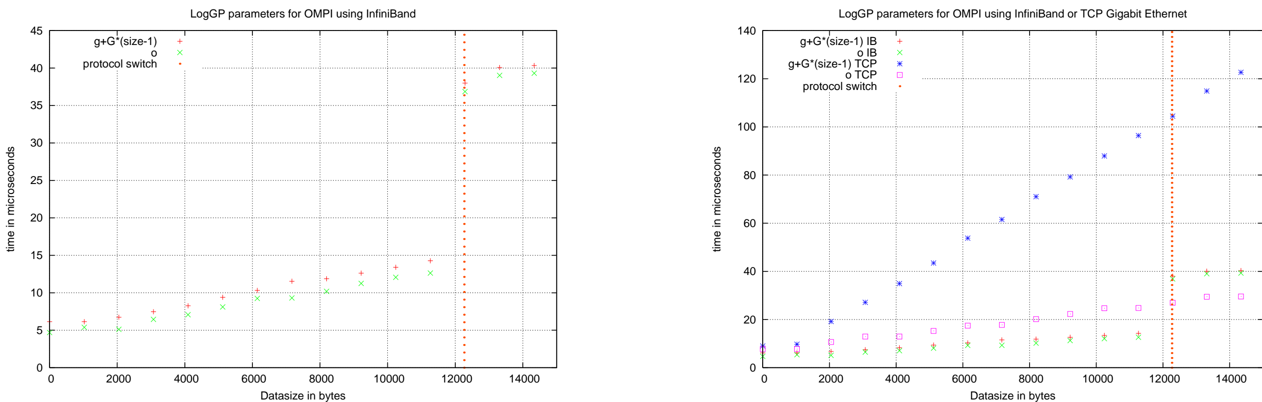


Figure 5.8.6.6: Some LogGP parameters measured using the Open MPI library. Right: for InfiniBand Left: for InfiniBand and TCP

When two InfiniBand interconnects used together, one could await that the resulting g,G values drop to around the half. For our current system this would mean that g,G drops below the o values. So the resulting bandwidth is limited by the ability of the library to process the messages. This is the reason, why the bandwidth does not double when we use two InfniBand interconnects together.

The o value of the TCP connection is higher than the g+G*(size-1) value. This means the libary will need more time to process a message when it is sent with TCP, then the system at a whole needs to process it. So for our system sending messages with TCP makes no sense when InfiniBand is available.

5.9 Overcoming the Bandwidth Limitations for Large Messages

Providing an efficient scheduling for large messages is only possible if the schedule is implemented in the PML and similar frameworks. Subsection 5.4 shows the difficulties with this approach. To get around this a pseudo bml btl module could be presented for every available BTL module in the RDMA list of the bml_base_endpoint object. Each pseudo bml btl module can hold the pointer to the Mpool module. This would allow the PML to do the scheduling for large messages and use RDMA again.

But this would mean that the virtual finishing time at each BTL module can not be adjusted. So the function pointers of the put operation have to be changed and have to point to a function of the FSBML component. This function can then adjust the VFT of the BTL module involved. This should lead to a comparable performance for large messages between the FSBML and the R2 component, but the benefits of the FSBML component when scheduling small messages are still available.

For get operation this approach would not be possible, because the operation is initiated at the receiving node. So the get operation would not be available. But this is no disadvantage because each available BTL component which provides the get operation also provides the put operation.

6. Conclusion and Further Works

We designed, implemented and tested a scheduling algorithm to schedule small messages. The algorithm was implemented in an own BML component of the Open MPI library with the name FSBML. We prove that the implementation does not introduce a significant overhead. For a given set of messages the used algorithm delivers a schedule which is very close to the optimum. It can handle different types of interconnects and so outperform the original implementation in such cases. The FSBML can also adapt to broken interconnects and situation when scheduling does not offer any advantages. At all it perform is in any cases as well as the original implementation and easily outperforms it when convenient conditions are given. So we proved that scheduling for small messages can provide performance gains and the goal to develop a prototype scheduler for small messages was achieved.
The performance of the FSBML depends on accurate information about the given interconnects. For this reason we used a well know model the LogGP model to describe the properties of interconnects. We had to realise that their was no accurate measurement method for the LogGP parameter available. So we also had to develop a own measurement method. The proposed algorithm was implemented in our FSBML module and now provides the information for scheduling. It has the advantage to be very fast and yet very accurate. It does not saturate the network so it seems promising to develop a further algorithm which could be used while run time. This would enable the application to detect changes in network performance because of contention. We are aware that the scheduling algorithm needs much more testing. Very important is that the performance of the FSBML component is tested with communication patterns using many nodes. Also it has to show that the FSBML perform well when message sizes and destinations are selected at random and not in such a regular manner like in the tests which were performed. Also the RDMA support has to be implemented in the FSBML. The algorithm was designed to schedule packets of arbitrary size so its efficiency for large messages has to be proven when RDMA support is implemented. Than it has to be shown that the FSBML improves the performance of real applications.

Because of time limitations we could not provide this tests at this work.
A main reason for the time problems is that the current architecture of Open MPI is not very good suited for scheduling. Many approaches were developed throughout this work and at the end were discarded. So developing a strategy how scheduling could be implemented in Open MPI acquired much more work then estimated.
A other reason is that we did not expected that it was so difficult to find an accurate model of network behaviour and a suited mean of measuring the parameters of the model. So much time was used to develop a accurate measurement method.
When it is proven, that the proposed algorithm works well in all above stated cases, then it should be more work done to improve the scheduling of large messages. The in subsection 5.9 proposed workaround could not provide optimal performance. Especially the problem of different nodes which could be reached with subsets of BTL modules has to be addressed. But this will mean huge changes at the architecture of Open MPI, and for that reason seems unlikely for the author.
If this work is done another issue could be addressed. The current architecture makes it impossible to identify BTL modules in different runs of the same application. If this would be possible it would be possible to save the LogGP parameter in a persistent memory. So measurement would be only necessary once for several runs of a application.
Another issue is the inability of the LogGP model to cover overhead which depends on message size. The LogGP model should be improved and the scheduling should be reviewed. This work may lead to a further performance improvement.
So this work is a first but promising step toward efficient scheduling of small messages, but still much work has to be done.

7. Appendix

7.1 References

[War98] Thomas Mike Warschko, Thomas Mike Karlsruhe: Effiziente Kommunikation in Parallelrechnerarchitekturen erschienen in Fortschr.-Ber. VDI Reihe 10: Informatik / Kommunikationstechnik Nr. 525 Düsseldorf: VDI Verlag 1998

[JMY89] Jesshope, C.R.; Miller, P.R.; Yantchev, JT.: High Performance Communications in Processor Networks. Procceedins International Symposium on Computer Architecture. (1989), S. 150-157

[NK93] Ni, L.M.; McKinley, P.K.: A Survey of Wormhole Routing Techniques in Direct Networks. IEEE Computer (1993) Nr. 2, S.62-76

[PJC96] Park, J.-Y.L.; Choi, H.-A.: Circuit Switched Broadcasting in Torus and Mesh Networks. IEEE Transactions on Parallel and Distributed Processing. Bd. 7 (1996) Nr. 2, S. 184-190

[BHP96] G. Bilardi, K.T. Herley, A.Pietracaprina: BSP vs LogP. In SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithmens and architectures, pages 25-32. ACM Press, 1996

[HK94] Susanne E. Hambrusch; Asfaq A. Khokhar: An architecture-independent model for coarse grained parallel machines. In Proceedings of the 6-th IEEE Symposium on Parallel and Distributed Processing, 1994

[Val90] Leslie G. Valiant. A bridging model for parallel computation. Commun. ACM, 33(8):103-111, 1990

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken: LogP: towards a realistic model of parallel computation. In Principles Practice of Parallel Programming, pages 1-12, 1993

[BKV00] Henri E. Bal; Thilo Kielmann; Kees Verstoep: Fast Measurement of LogP Parameters for Message Passing Platforms. In Lecture Notes In Computer Science; Vol. 1800 Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing Pages: 1176 – 1183, 2000

[AIS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman: LogGP: Incorporating Long Messages into the LogP Model. Journal of Parallel and Distributed Computing, 44(1):71-79, 1995

[BHP05] Gianfranco Bilardi; Kieran T. Herley, Andrea Pietracaprina, and Geppino Pucci: On stalling in LogP. Journal of Parallel and Distributed Computing, Volume 65 , Issue 3 Pages: 307 – 312, Academic Press, Inc., 2005

[IFH01] Fumihiko Ino, Noriyuki Fujimoto and Kenichi Hagihara: LogGPS: A Parallel Computational Model for Synchronization Analysis. Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming PPoPP '01, Pages: 133 – 142, ACM Press, 2001

[MF01] Csaba Andras Moritz; Matthew I. Frank: LoGPC: Modeling Network Contention in Message-Passing Programs. IEEE Transactions on Parallel and Distributed Systems, 12(4) , Pages: 404 – 415, 2001

[FAV97] Matthew I. Frank; Anant Agarwal; Mary K. Vernon: LoPC: modeling contention in parallel algorithms. Principles and Practice of Parallel Programming Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming, Pages: 276 – 287, ACM Press, 1997

[Sch98] Scheideler, Christian: Universial routing strategies for interconnection networks. Lecture Notes in Computer Science 1390, Springer, 1998

[YD98] Yalamanchili, Sudhakar; Duato, José: Parallel Computer Routing and Communication: second international workshop. Lecture Notes in Computer Science 1417, Springer, 1998

[HP93] Harrison, Peter G.; Patel, Naresh M.: Performance modelling of communication networks and computer architectures, Addison-Wesley Publishers Ltd., 1993

[H94] R.W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Computing, 20(3):389–398, March 1994.

[Pap99] Lothar Papula; Mathematik für Ingenieure und Naturwissenschaftler Band 3: Vektoranalysis Wahrscheinlichkeitsrechnung Mathematische Statistik Fehler- und Ausgeichsrechnung, 3. Auflage, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1999

[Lic88] William Lichten; Data and Error Analysis in the Intrductory Physics Laboratory, Allyn and Bacon Inc., Newton, 1988

[CLMY96] David Culler, Lok Tin Liu, Richard P. Martin, Chad Yoshikawa; LogP Performance Assessment of Fast Network Interfaces: IEEE Micro, 16(1):35-43, Feb. 1996.

[RHR06] Mirko Reinhardt, Torsten Höfler, Wolfgang Rehm; Optimizing Point-to-Point Ethernet Cluster Communication, Diploma Thesis: Faculty of Computer Science, Chemnitz University of Technologie, February 2006.

[MPI95] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. 1995.

[MPI97] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997.

[ILM98] G. Iannello, M. Lauria, and S. Mercolino. Logp performance characterization of fast messages atop myrinet, 1998.

[BBC03] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An evaluation of current highperformance networks. In IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, page 28.1, Washington, DC, USA, 2003. IEEE Computer Society.

[Pal00] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, Pallas GmbH, 2000.

[NVZ01] J. Nieh, C. Vaill, H. Zhong, "Virtual-time round-robin: An O(1) proportional share scheduler," in Proceedings of the 2001 USENIX Annual Technical Conference, USENIX, Berkeley, CA, June 25–30 2001, pp. 245–259

[CCN05] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, Haoqiang Zheng. Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems

[LVP04] Jiuxing Liu, Abhinav Vishnu, D. K. Panda. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation, Proceedings of the ACM/IEEE SC2004 Conference, Nov. 2004

[Net06] Netgauge. http://www.unixer.de/researche/netgauge/, 2006.

[GFB04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall; Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation: Proceedings, 11th European PVM/MPI Users' Group Meeting, Sep. 2004

[CWD05] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, G .E. Fagg}.The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing, in Proceedings, 12th European PVM/MPI Users' Group Meeting, 2005, Sorrento, Italy, Sep.

[GSB06] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, Andrew Lumsdaine. Open MPI: A High-Performance, Heterogeneous MPI, in Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2006, Barcelona, Spain, Sep.

[SL04] Jeffrey M.\ Squyres, Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms, Proceedings, 18th ACM International Conference on Supercomputing, Workshop on

Component Models and Systems for Grid Applications, pages 167—185, 2004, St.\ Malo, France, Jul, Springer

[SWB06] Galen M. Shipman, Tim S. Woodall, George Bosilc, Arthur B. Maccabe. High Performance RDMA Protocols in HPC ,Euro PVM/MPI 2006, September, 2006, in Bonn, Germany.

[BBG06] Brian Barrett, George Bosilca, Rich Graham, Galen Shipman, Tim Woodall, Jeff Squyres. Open MPI Developer's Workshop, Presented: April 17-20, 2006, Cisco Systems Campus, San Jose, CA, USA.http://www.open-mpi.org/papers/workshop-2006/

7.2 List of Figures

Figure 2.2.7.1: Transmission scheme to to nodes in: upper left PLogP; upper right LogGP; down LogGP....11
Figure 2.2.7.2: Transmission scheme to two nodes in: upper left PLogP; upper right LogP; down LogGP (5 bytes)....11
Table 2.2.12.1: Classification of network models....12
Figure 3.2.2.1: PRTT for 1 5 byte packet: green the total time for one transmission....18
Figure 3.2.2.2: PRTT for 3 5 byte packets without delay: red the RTT for one packet; green the time for g+(5-1)*G; gray the part without influence on the result time....18
Figure 3.2.2.3: The PRTT for 3 5 byte packets with a delay of d: red the RRT for one packet; green the time of o+d; gray parts that do not influence the result....19
Figure 3.5.2.1: round trip time InfiniBand....25
Figure 3.5.2.2: round trip time Myrinet/GM....25
Figure 3.5.2.3: round trip time Ethernet TCP....25
Figure 3.5.2.4: PRTT_16_d InfiniBand....25
Figure 3.5.2.5: PRTT_16_d Myrinet/GM....25
Figure 3.5.2.6: PRTT_16_d Ethernet TCP....25
Figure 3.5.3.1: g,G InfiniBand....26
Figure 3.5.3.2: g,G Myrinet/GM....26
Figure 3.5.3.3: g,G Ethernet TCP....26
Figure 3.5.3.4: o InfiniBand....27
Figure 3.5.3.5: o Myrinet/GM....27
Figure 3.5.3.6: o Ethernet TCP....27
Figure 5.2.1: A MCA framework example....41
Figure 5.5.2.1: Class dependencies between the PML and BML....44
Figure 5.5.3.1: Overview of important BML data structures....45
Figure 5.6.1.1: Scheduling information....48
Figure 5.6.2.1: fsbml layout at each process; green: the extensions....49
Figure 5.8.3.1: Open MPI using one InfiniBand interconnect....53
Figure 5.8.3.2: Open MPI using one Myrinet/GM interconnect....53
Figure 5.8.3.3: Open MPI using TCP with one available Gigabit Ethernet interconnect....54
Figure 5.8.4.1: Open MPI performance using two InfiniBand interconnects....54
Figure 5.8.4.2: Open MPI performance using TCP across two Gigabit Ethernet interconnects....55
Figure 5.8.5.1: Open MPI performance using one InfiniBand and one TCP Gigabit Ethernet interconnect....55
Figure 5.8.5.2: Open MPI performance using one Myrinet/GM and one broken TCP Gigabit interconnect....55
Figure 5.8.5.3: Open MPI performance using TCP across one Gigabit Ethernet interconnect and TCP across one Fast Ethernet interconnect....56
Figure 5.8.6.1: Open MPI using two InfiniBand interconnects compared to using one....56
Figure 5.8.6.2: OMPI using two TCP Gigabit Ethernet connections compared to using one....57

Figure 5.8.6.3: Open MPI using InfiniBand together with TCP across Gigabit Ethernet compared to using one single InfiniBand interconnect.....58
Figure 5.8.6.4: Open MPI using Myrinet/GM together with TCP across Gigabit Ethernet compared to using one single Myrinet/GM interconnect.....58
Figure 5.8.6.5: Open MPI using TCP across one Gigabit Ethernet interconnect together with TCP across one Fast Ethernet interconect compared to using one single Gigabit Ethernet interconnect. .59
Figure 5.8.6.6: Some LogGP parameters measured using the Open MPI library. Right: for InfiniBand Left: for InfiniBand and TCP.....59

7.3 List of tables

Table 3.5.1.1: Details about the test systems.....24
Table 3.5.3.1: The LogGP parameter (all values are microseconds).....28
Table 5.8.2.1: Details about the test systems.....52

7.4 List of Abbreviations

API	Application Programming Interface
BML	BTL Management Layer
BSP	Bulk Synchronous Protocol
BTL	Binary Transmission Layer
CPU	Central Processing Unit
EEL	End to End Latency
HPC	High Performance Computer
I/O bus	Input / Output bus
MCA	Modular Component Architecture
MPI	Message Passing Interface
NIC	Network Interface Card
OPAL	Open Portability Abstraction Layer
OSC	One Side Commutation Layer
PML	Point-to-Point Management Layer
PC	Personal Computer
PRTT	Parametrized Round Trip Time
RDMA	Remote Direct Memory Access
RTE	Run Time Environment
RTT	Round Trip Time
SMP	Symmetric Multiprocessor systems
TCP	Transmission Control Protocol
VFT	Virtual Finishing Time

7.5 List of Trademarks

Athlon	Trademark of Advanced Micro Devices
InfiniBand	Trademark of the InfiniBand Trade Association
InfiniHost	Trademark of Mellanox
Opteron	Trademark of Advanced Micro Devices
Myrinet	Trademark of Myricom Inc.

The above list names trademarks or service marks that are used in this document and are hold by their respectiv owners in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

7.6 Acknowledgements

I want to thank my advisers Torsten Höfler and Prof. Dr. Rehm for their technical support, for many inspiring discussion and useful hints. I also want to thank Frank Mietke for the administration and support of the testing environment and Torsten Mehlan for many useful hints according programming in Open MPI. I have to thank Charlemagne Burt, who spend many hours voluntary to improve the language of my theses. I also want to thank my parents who supported me during this work and keep away most of the normal life troubles. At last but not least I want to thank my friends who always cheered me up when work become frustrating or problems seems unsolvable.

Statutory Declaration

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, no other sources or auxiliary tools except those stated, referenced and acknowledged have been used.

Chemnitz, November 02, 2006
André Lichei