written and implemented at

Faculty of sciences
Department of computer science
Vrije Universiteit Amsterdam
The Netherlands

submitted to

Fachbereich 5
Wirtschaftswissenschaften
Universität Siegen
Germany

A Flexible and Efficient Message Passing Platform for Java

Eine flexible und effiziente Message Passing Plattform für Java

Markus Bornemann

A Thesis presented for the Degree of
Diplom-Wirtschaftsinformatiker

Author:	Markus Bornemann
Student register:	542144
Address:	Amselweg 7
	57392 Schmallenberg
	Germany

Supervisor:	Dr.-Ing. habil. Thilo Kielmann
Second reader:	Prof. Dr. Roland Wismüller

Amsterdam, September 2005

Zusammenfassung

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Problems which are too complex to be solved by theory or too cost-intensive for practical approaches can only be handled using simulation models. Some of these problems, for example genetic sequence analysis or global climate models, getting to large to be solved on a single processor machine in reasonable time or simply exceed the physical limitations provided.

Usually, parallel computers providing a homogeneous networking infrastructure had been used to address these problems, but in times of low budgets more and more existing local area networks integrate multiple workstations or PCs making them suitable for parallel computing. In contrast to computer clusters these machines typically consist of different hardware architectures and operating systems.

The Message Passing Interface (MPI) developed by the MPI-Forum has been widely accepted as a standard for parallel computation on high performance computer clusters. Since MPI is widely used and a lot of experience has been built up, there is a growth interest of making the MPI concepts applicable for heterogeneous systems. Unfortunately, existing MPI implementations written mostly in C and Fortran are limited to the hardware architecture they are implemented for. In the past portability was of minor importance. Therefore, MPI applications need to be adapted and recompiled while changing the target platform.

During the last years the Java programming language has become very important for application developers, when heterogeneous, networking environments need to be addressed. Java allows to develop applications that can run on a variety of different

computer architectures and operating systems without recompilation. Additionally, Java has been designed to be object-oriented, dynamic and multi-threaded.

Nowadays, performance of hardware and connectivity has rapidly increased allowing to use heterogeneous infrastructures of for instance small and medium sized companies or the Internet for parallel computation. Therefore, parallel applications need to match the requirements of being portable and flexible allowing them to be executed on different architectures simultaneously without the high costs of porting software explicitly. Naturally, the ability of being flexible should not come with significant performance drawbacks.

In this thesis, a message passing platform called MPJ/Ibis is presented that combines both efficiency and flexibility. Chapter 2 gives an overview of parallel architectures in general and introduces the basic concepts of MPI. Then, Chapter 3 points out Javas abilities and limitations for parallel computing and describes the grid programming environment Ibis, which addresses Javas drawbacks. The main design and implementation details of MPJ/Ibis are described in Chapter 4 followed by a presentation of various benchmark results in Chapter 5. Finally, in Chapter 6, previous projects concerning message passing for Java are introduced with a view to flexibility and efficiency.

Chapter 2

The Message Passing Interface

The Message Passing Interface (MPI) is a standardized message passing model and not a specific implementation or product. It was designed to provide access to advanced parallel hardware for end users, library writers and tool developers. MPI was standardized by the MPI-Forum in 1995 (MPI-1.1) and further developed to MPI-2, which includes MPI-1.2, in the following years. The existence of a standardized interface makes it convenient to software developers implementing portable and parallel programs with the guaranteed functionality of a set of basic primitives. It defines the general requirements for a message passing implementation, presenting detailed bindings to C and Fortran languages as well.

This introduction to MPI refers to MPI-1.1 [15] and does not claim to be a complete reference or tutorial, rather it explains the basic principles for the implementation of a message passing platform for Java. Therefore the author avoids, where it is possible, the presentation of MPI function bindings.

2.1 Parallel Architectures

A parallel machine consists of a various number of processors, which collectively solve a given problem. In contrast to single processor machines, e. g. workstations, parallel machines are able to execute multiple instructions simultaneously. This results in the main motivation of parallel computing to solve problems faster.

To describe parallel computer architectures in general, Flynn's taxonomy [24,

640] characterizes four different classes of architectures. It uses the concept of streams, which at least are sequences of items processed by a CPU. A stream consists either of instructions or of data, which will be manipulated by the instructions. These are:

SISD	Single instruction, single data streams
MISD	Multiple instructions, single data streams
SIMD	Single instruction, multiple data streams
MIMD	Multiple instructions, multiple data streams

Table 2.1: Flynn's Taxonomy (after [24, 640])

The first item, SISD, describes a sequential architecture, in which a single processor operates on a single instruction stream and stores data in a single memory, e.g. a von Neumann architecture. SISD does not address any parallelization in the mentioned streams. MISD is more or less a theoretical architecture, where multiple instructions operate on single data streams simultaneously. In fact multiple instruction streams need multiple data streams to be effective, therefore no commercial machine exists with this design. Computers working on the SIMD model manipulate multiple data streams with the same set of instructions. Usually this will be done in parallel, e.g. in an array processor system. In this model all processors are being synchronized by a global clock to make sure that every processor is performing the same instruction in lockstep. MIMD concerns about fully autonomous processors, which perform different instructions on different data. This case implies that the computation has to be done asynchronously.

Furthermore, parallel systems, i.e. SIMDs and MIMDs, differ in the way the processors are connected and thus how they communicate with each other. On the one hand all processors may be assigned to one global memory, called shared memory. On the other hand every processor may address its own local memory, called distributed memory. All the paradigms above regard to hardware. In addition to the above, there is a software equivalent to SIMD, called SPMD (Single program, multiple data). In contrast to SIMD, SPMD works asynchronously, where the same program runs on processors of a MIMD system.

The message passing concept matches the SPMD paradigm, where communica-

tion takes place by exchanging messages. In the message passing world programs consist of separate processes. Each process addresses its own memory space, which will be managed by the user as well as data distribution among certain processes.

2.2 MPI Concepts

MPI is based on a static process model, that means all processes within an existing MPI runtime environment enter and exit this environment simultaneously. Inside the running environment all the processes are being organized in groups. Actually the communication occurs by accessing communicators, group defined communication channels. Furthermore, the processes inside a group are numbered from 0 to `n`-1, where `n` is the total number of processes within the group. Those numbers are called ranks. For all members of the group the rank number of a certain member is the same. That allows a global view to the group members.

Message Data Assuming a process is about to send data, it has to specify what data is going to be sent. The location of this data is called the send buffer. On the other side, when data has to be received, this location is called the receive buffer.

Messages may consist of contiguous data, e.g. an array of integer values. In order to avoid memory-to-memory copying, e.g. if just a smaller part of an array inside the send buffer should be transfered, the number of elements has to be specified for a message, so that the needed elements may be used directly.

Since MPI could be implemented as a library, which may be used precompiled, it cannot be assumed that a communication call has information about the datatype of variables in the communication buffer. Hence, MPI defines its own datatypes, which then will be attached to the message. A reduced list for the C language binding is presented in Table 2.2.

A clearer reason to attach datatypes explicitly to messages is shown by the case where non-contiguous data will be submitted, e.g. a column of a two-dimensional array. For that, the user has to construct a type map, which specifies the needed elements, combined with a MPI datatype. This results in a derived datatype based on MPI datatypes.

MPI Datatypes	C Datatypes
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
...	...

Table 2.2: MPI datatypes and corresponding C types

Message Envelopes To realize communication in general, MPI uses the so-called message envelopes. Envelopes are meant to describe the necessary details of a message submission. These are:

- Sender
- Receiver
- Tag
- Communicator

The sender and receiver items just hold information about the ranks of the communication partners, whereas the used group will be identified by the communicator. The tag is a free interpretable positive integer value. It may be used to distinguish different message types inside the communicator.

2.3 Point-to-Point Communication

Communication between exactly two processes is called point-to-point communication in MPI. In this way of communication both the sender and receiver are explicitly recognized by the underlying communicator and therefore the specific ranks inside the process group. MPI defines two possibilities to achieve point-to-point communication, blocking and non-blocking, which will be explained in Sections 2.3.1 and 2.3.2. Besides that there are three different communication modes.

First, in *ready send mode* the message will be sent as the receiver side the called matching receive function. It has to be called before a sender process is able to send

a message, otherwise this mode results in error. Second, in *synchronous send mode*, the sending process first requests for a matching receive function on the receiver side, and then, if a matching receive has been posted, sends the message. Third, it is possible to buffer the message before it will be sent. This mode is called *buffered send*, where the message first will be copied into a user defined system buffer. The real communication then may be separated from the sending process and moved to the runtime environment. On the receiver side these communication modes appear fully transparent. The receiver process doesn't have any influence on the communication mode used.

2.3.1 Blocking Communication

Blocking communication function calls return when the operation assigned to the function has finished. It blocks the caller until the involved buffer may be reused.

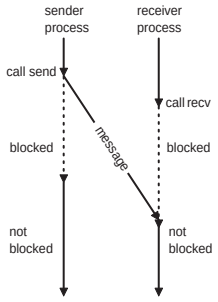

Figure 2.1: Blocking Communication Demonstration

Figure 2.1 demonstrates a blocking send and the associated receive process. The sending process calls send and then returns when the message has left the buffer completely. A finished send call does not imply that the message has arrived at it's destination entirely, whereas the receiver blocks until the message has been arrived completely.

2.3.2 Non-Blocking Communication

In contrast to blocking communication the operations of the non-blocking point-to-point communication return immediately. That allows a process to do further

computation during message transfer. On the other hand it is not possible to use the involved buffers on the sender and receiver side until the point-to-point transfer has been finished. To figure out when a communication is done, MPI provides wait and test functions, allowing a process to check for a message transfer to complete (see figure 2.2).

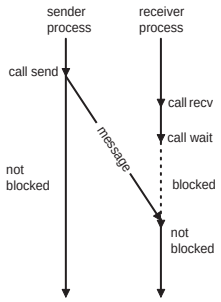

Figure 2.2: Non-Blocking Communication Demonstration

2.4 Collective Communication

On top of the point-to-point communication, MPI provides functions to communicate within groups of processes, called collective communication.

Collective operations are executed by all members of the group. These operations are responsible for synchronization, broadcasting, gathering, scattering and reducing of information between the groups of processes. Some operations need a special process to send data to, or collect data from, the other processes, which is called the root process. A collective operation always blocks the caller until all processes have called it. In the following, the most important collective operations will be introduced.

Broadcast Figure 2.3 illustrates a *broadcast* of six processes. The left table shows the initial state of the buffers of all processes, each row for a process. The first process is the root, which sends its input buffer to all processes including itself. In the end all processes hold a copy of the root's input buffer, as shown on the right side of Figure 2.3.

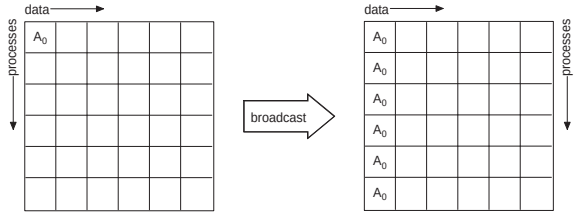

Figure 2.3: Broadcast Illustration

Reduce The *reduce* operation combines all items of the send buffer of each process using an associative mathematical operation. The result then will be sent to the root process. Figure 2.4 demonstrates reduce, where the result appears in R_0. MPI defines a set of standard operations for that purpose, which are listed in table 2.3.

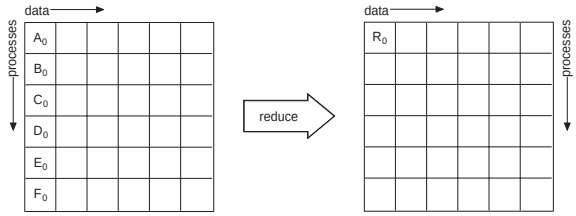

Figure 2.4: Reduce Illustration

Name	Meaning
MPI_MAX	maximum value
MPI_MIN	minimum value
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Table 2.3: Predefined Reduce Operations

Of course there are restrictions of which datatypes are accepted by the different operations. For example it does not make sense to calculate a bit-wise or on a set of

floating point values. All predefined reduce operations are commutative. In addition to that, MPI allows to create user defined reduce operations, which may or may not be commutative. Therefore a MPI implementation should take care about the way the reduce will be performed in order to respect non-commutative behaviour. An extension to reduce is the allreduce operation where all processes receive the result, instead of just root.

Scatter/Gather In *scatter* mode, the send buffer of root will be split into n equal parts, where n is the number of processes inside the group. Each process then receives a distinct part, which will be determined by the rank number of the receiving process.

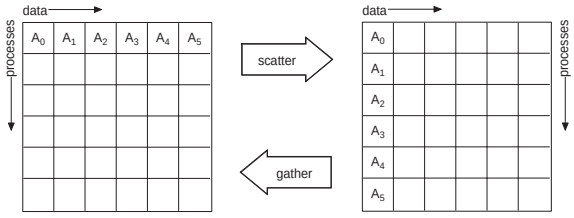

Figure 2.5: Scatter/Gather Illustration

Gather is the inverse operation of scatter. All processes send their send buffers to root, where all the incoming messages will be stored in rank order into root's receive buffer.

Allgather An extension to *gather* is *allgather*. As shown in figure 2.6 instead of just root, all processes of the group receive the gathered data.

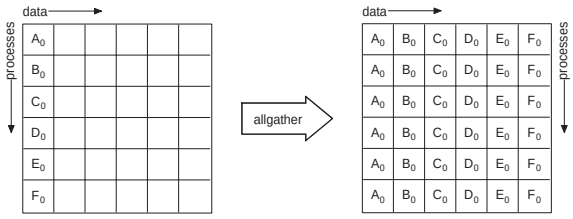

Figure 2.6: Allgather Illustration

Alltoall The last extension is *alltoall*. In *alltoall* each process sends distinct data to each other process. This means that the `j`th item sent from process `i` is received by process `j` in the `i`th place of the receive buffer, where `i` and `j` are rank numbers of group processes.

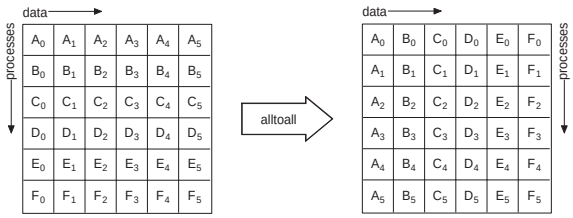

Figure 2.7: Alltoall Illustration

Other Functions In addition to the collective operations mentioned above, MPI provides more functions. It is beyond the objective of this thesis to explain all of them at this point. Only two further collective operations will be described in the following.

MPI allows a group of processes to synchronize on each other. To achieve synchronization each process has to call the collective function *barrier*. When a process calls the *barrier*, it blocks until all the other processes have called the *barrier* as well.

A variant to *allreduce* is the operation *scan*. It performs a prefix reduction, where process `i` receives the reduction of the items from the processes `0, ..., i`.

2.5 Groups, Contexts and Communicators

Initially MPI creates a group, in which all involved processes of the runtime environment are listed. Beneath that, it provides the ability to generate other process groups, in order to help the programmer to achieve a better structure of the source code. This is important, e.g. for library developers, to focus on certain processes in collective operations to avoid synchronizing or running unrelated code on uninvolved processes.

The rank of a process is always bound to a certain group, which means that

a process that is part or several different groups may have different ranks inside those groups. Therefore it is necessary to obtain knowledge about a group, before communication can take place. This is done via the communicators. Each group owns at least one communicator, with what the group members are able to deliver messages to each other. Each communicator is assigned to strictly one group and the relationship between groups and communicators creates the context for the processes.

Groups Each process belongs at least to one group, namely the initially created group represented by the communicator *MPI_ COMM_ WORLD*, which contains all processes. MPI itself does not provide a function for an explicit construction of a group, instead a new group will be produced by using reduction and mathematical set operations on existing groups, which result into a new group including the specified or calculated processes. The set operations are listed explicitly in Table 2.4, where it will be assumed that the operations are being executed on two different groups as arguments, called `group1` and `group2`.

Operation	Meaning
union	all processes of `group1`, followed by all processes of `group2` not in `group1`
intersection	all processes of `group1` that are also in the `group2`, ordered as in `group1`
difference	all processes of `group1` that are not in `group2`, ordered as in `group1`

Table 2.4: Group Construction Set Operations

As mentioned above, it is also possible to reduce an existing group, using the so-called functions incl and excl. In both the user has to specify certain or a range of ranks, which will be included or excluded from the existing group, respectively.

Intra-Communicators Communicators always occur in relationship with groups. Since processes just send and receive messages over communicators and because communicators represent the channels within groups, the set of communicators assigned to a process forms the closure of the systems capability of communication. Each

communicator is assigned to exactly one group and each group belongs to at least one communicator. For example two processes, each is part of a different group, cannot exchange messages directly. In order to achieve this possibility, a new group must be created with a new communicator attached. Therefore MPI provides functions to construct communicators from existing groups. Communicators concerning communication within groups are called intra-communicators.

Inter-Communicators As shown above, creating new groups and communicators is quite inconvenient just for the purpose that processes from different groups want to exchange messages. For this special case MPI specifies inter-communicators. Those communicators are being constructed by specifying two intra-communicators, between which the inter-communication takes place.

2.6 Virtual Topologies

The previous introduced intra-communicators address a linear name space, where the processes are being numbered from 0 to `n`-1 (see Section 2.2). In some cases such a numbering does not reflect the logical communication structure. Depending on the underlying algorithm, structures like hypercubes, n-dimensional meshes, rings or common graphs appear. These logical structures are called the virtual topology in MPI, which allows an additional arrangement of the group members. Since the communication operations identify source and destination by ranks, a virtual topology calculates, for instance, the original rank to the neighbour of a certain process or a process with specified coordinates. A virtual topology is an optional attribute to intra-communicators and will be created by corresponding MPI functions, while inter-communicators are not allowed to use virtual topologies.

Graph Topologies Each topology may be represented by a graph, in which the nodes represent the processes and the edges represent the connections between them. A missing edge between two nodes doesn't mean that the two processes aren't able to communicate to each other, rather the virtual topology simply neglects it. The edges are not weighted. It is that there is a connection between two nodes or not.

In figure 2.8 two examples, a binary tree with seven processes and a ring with eight processes, are shown.

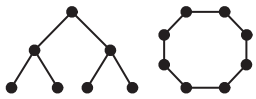

Figure 2.8: Graph Topologies

Cartesian Topologies Cartesian structures are simpler to specify than common graphs, because of their regularity. Even if a cartesian raster may be described by a graph as well, MPI provides special functions to create those structures in order to give more convenience to the user. Figure 2.9 shows examples of a 3-dimensional mesh with eight processes and a 2-dimensional mesh with nine processes.

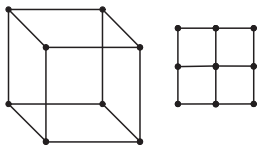

Figure 2.9: Cartesian Topologies

Chapter 3

The Grid Programming Environment Ibis

With the concept "Write once, run everywhere", Java provides a solution to implement highly portable programs. Therefore Java source code will not be compiled to native executables, but to a platform independent presentation, called bytecode. At runtime the bytecode will be interpreted or compiled just in time by a Java Virtual Machine (JVM). This property made Java attractive, especially for grid computing, where many heterogeneous platforms are used and where portability becomes an issue with compiled languages. It has been shown, that Javas execution speed is competitive to other languages like C or Fortran [3].

In the following, Javas abilities related to parallel programming will be pointed out and then the grid programming environment Ibis and its enhancements will be introduced.

3.1 Parallel Programming in Java

In order to achieve parallel programming in general, Java offers two different models out of the box. These are:

1. Multithreading for shared memory architectures and
2. Remote Method Invocation (RMI) for distributed memory architectures.

The RMI model enables almost transparent communication between different JVMs and was at first the main point of interest for research on high-performance computation in Java.

3.1.1 Threads and Synchronization

Concurrency has been integrated directly into Java's language specification using threads [13, 221-250]. A thread is a program fragment, that can run simultaneously to other threads similar to processes. While a process is responsible for the execution of a whole program, multiple threads could run inside that process. The main difference between threads and processes is, that threads are sharing the same memory address space, while the address spaces of processes are strictly separated.

To create a new thread, an object of a class, which extends *java.lang.Thread,* has to be instantiated. Since Java does not support multiple inheritance, there is also an interface called *java.lang.Runnable*, that allows a class that is already derived from another class to behave as a thread. This object will be committed to the constructor of an extra created *java.lang.Thread* object. In both cases a method called *run()* has to be implemented, which is the entry point when invoking a thread. Invoking will be done by calling the method *start()*.

Because threads run in the same address space and thus share the same variables and objects, Java provides the concept of monitors to prevent data from being overwritten by other threads. All objects of the class *java.lang.Object,* and therefore all objects of classes derived from *java.lang.Object* in contrast to primitive types, contain a so-called lock. A lock will be assigned to exactly one thread, while other threads have to wait until the lock has been released. With that mutual exclusive lock (short: mutex lock) it is possible to coordinate access to objects and thus avoiding conflicts. Java doesn't allow direct access to locks. Instead the keyword *synchronized* exist to label critical regions. By using *synchronized* it is possible to protect a whole method or a fragment within a method. When applying *synchronized* on a whole method, *this* pointer's lock will be used, otherwise an object lock is used by passing the object reference to it.

In addition to monitors, Java supports conditional synchronization providing

the methods *wait()* and *notify()* (or *notifyAll()*). Both methods may only be called when the associated thread is the owner of the object's lock. A call to *wait()* informs the thread to wait until another thread invokes *notify()*. A more specific method is *join()*, which blocks the current thread until the associated thread has finished entirely. Figure 3.1 demonstrates the wait and notify model by showing two threads that have been synchronized on an object. Thread 1 sets a value to the object and waits until Thread 2 notifies it, when the value has been collected. After notification Thread 2 sets another value to the object.

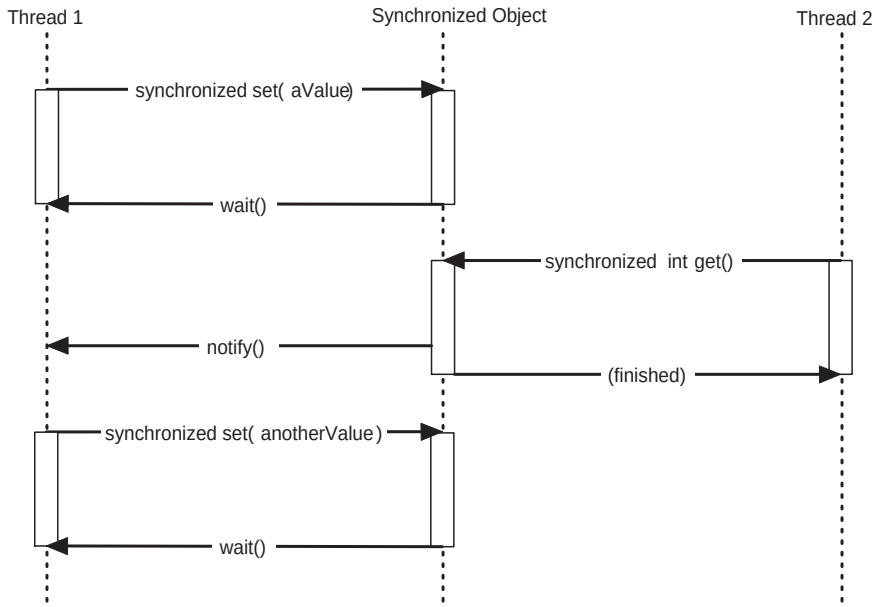

Figure 3.1: Java thread synchronization example

3.1.2 Remote Method Invocation

An interesting way to address distributed architectures is the Remote Method Invocation API [22] provided by Java since JDK 1.1. RMI extends normal Java programs with the ability to share objects over multiple JVMs. It doesn't matter where the JVMs are located and therefore RMI programs work in heterogeneous environments. RMI in principle is based on a client/server-model, where a remote object, which should be accessible from other JVMs, is located on the server side. On the client side, a remote reference to that object appears which allows the client to invoke methods of the remote object. Internally remote objects can be collected in the

RMI registry. Figure 3.2 shows two JVMs demonstrating the method invocation.

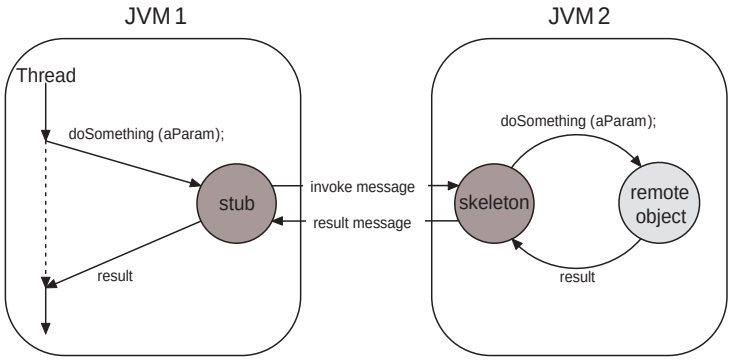

Figure 3.2: RMI invocation example (after [14, 12])

When looking up a remote object the client gets a stub, which implements the accessors to the remote object and acts like a usual Java object. The counterpart on the server side is called the skeleton. Both will be generated by the RMI compiler. By invoking a remote method the stub marshalls the invocation including the method's arguments and sends it to the skeleton, where it will be unmarshalled and forwarded to the remote object. The result from the remote object will be submitted in the same manner.

The communication between stub and skeleton always via TCP/IP, is synchronous. Therefore a stub always has to wait until the invocation on the remote object has finished. The great advantage of RMI is the fact, that it abstracts completely from low-level socket programming. That makes it more convenient to software developers to distribute applications in Java. On the other hand RMI shows dramatic performance bottlenecks, since it uses Java's object serialization [23] and reflection mechanism for data marshalling. It has been evaluated [14, 11-36] that RMIs communication overhead can result in high latencies and low throughput, which in the end does not result in significant speedups for parallel applications, in fact some applications can become slower [14, 34].

3.2 Ibis Design

As shown above, Java natively does not provide solutions to achieve portable and efficient communication for distributed memory architectures. In the following, the grid programming environment Ibis[1] [26] will be introduced, which addresses portability, efficiency and flexibility. Ibis has been implemented in pure Java. However in special cases some native libraries are using the Java Native Interface, which can be used to improve performance.

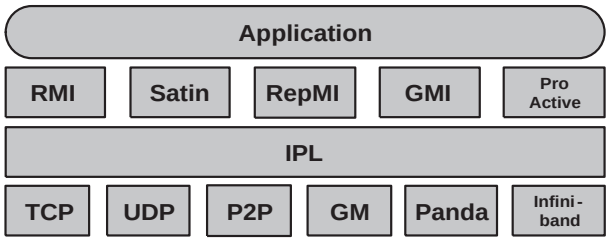

Figure 3.3: Ibis design (redraw from the Ibis 1.1 release)

The main part of Ibis is the Ibis Portability Layer (IPL). It provides several simple interfaces, which are implemented by the lower layers (TCP, UDP, GM, ...). These implementations can be selected and loaded by the application at run time. For that purpose Ibis uses Java's dynamic class loader. With that applications can run simultaneously on a variety of different machines, using optimized and specialized software where possible (e.g. Myrinet) or standard software (e.g. TCP) when necessary. Ibis applications can be deployed on machines ranging from clusters with local, high-performance networks like Myrinet or Infiniband, to grid platforms in which several, remote machines communicate across the Internet. RMI (see 3.1.2) does not support these features. Although it is possible, Ibis applications will typically not be implemented on top of the IPL. Instead they use one of the existing programming models. One of these models is a reimplementation of RMI, that allows a comparison between Ibis and the original RMI [25, 169-171] and shows Ibis' advantages. The other models will not be discussed here. Figure 3.3 shows the layered structure of the current Ibis release (Version 1.1).

[1] Ibis online at http://www.cs.vu.nl/ibis/

Send and receive ports To enable communication, the IPL defines send and receive ports (Figure 3.4), which provide a unidirectional message channel. These ports must be connected to each other (connection oriented scheme). The communication starts by requesting a new message object from the send port, where data items of any type, even objects, of any size can be inserted. After insertion, the message will be submitted by invoking *send()*.

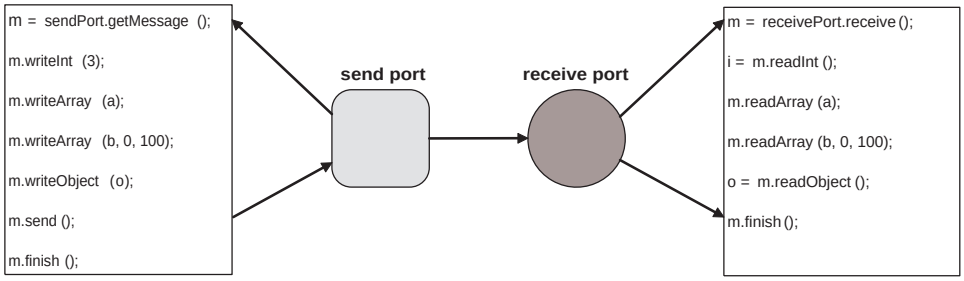

Figure 3.4: Send and Receive Ports (after [25, 153])

Each receive port may be configured in two different ways. Firstly, messages can be received explicitly by calling the *receive()* primitive. This method is blocking and returns a message object from which the sent data can be extracted by the provided set of read methods (see Figure 3.4). Secondly, Ibis achieves implicit receipt with the ability to configure receive ports to generate upcalls. If an upcall takes place, a message object will be returned. These are the only communication primitives that the IPL provides. All other patterns can be built on top of it.

3.3 Ibis Implementations

Besides flexibility, Ibis provides two important enhancements: Efficient serialization and communication [25, 165-169]. The message passing implementation, which will be introduced in Chapter 4, takes advantage of both.

Efficient serialization As shown in Section 3.1.2 Java's object serialization is a performance bottleneck. Ibis circumvents it by implementing it's own serialization mechanism that is fully source compatible to the original. In general, Ibis serialization achieves performance advantages in three steps:

- Avoiding run time type inspection
- Optimizing object creation
- Avoiding data copying

This has been done by implementing a bytecode rewriter, which adds a specialized generator class to all objects extending the serializable interface and takes over the standard serialization. Evaluations [25, 164] have pointed out that Ibis serialization outperforms the standard Java serialization by a large margin, particularly in those cases where objects are being serialized.

Efficient communication The TCP/IP Ibis implementation is using one socket per unidirectional channel between a single send and receive port which is kept open between individual messages. The TCP implementation of Ibis is written in pure Java allowing to compile an Ibis application on a workstation, and to deploy it directly on a grid. To speedup wide-area communication, Ibis can transparently use multiple TCP streams in parallel for a single port. Finally, it can communicate through firewalls, even without explicitly opened ports.

There are two Myrinet[2] implementations of the IPL, built on top of the native GM[3] (Glenn's messages) library and the Panda [2] library. Ibis offers highly-efficient object serialization that first serializes objects into a set of arrays of primitive types. For each send operation, the arrays to be sent are handed as a message fragment to GM, which sends the data out without copying. On the receiving side, the typed fields are received into pre-allocated buffers; no other copies need to be made.

[2]Myrinet online at http://www.myri.com/
[3]Myrinet GM driver online at http://www.myri.com/scs/

Chapter 4

Design and Implementation of MPJ/Ibis

The driving force in high performance computing for Java was the Java Grande Forum[1] (JGF) from 1999 to 2003. Before Java Grande many MPI-like environments in Java were created without declaring any standards. Thus, a working group from the JGF proposed a MPI-like standard description, with the goal to find a consensus API for future message-passing environments in Java [4]. To avoid naming conflicts with MPI, the proposal is called Message Passing Interface for Java (MPJ).

4.1 Common Design Space and Decisions

Since MPI is the de facto standard for message passing platforms and MPJ is mainly derived from it, the decision to create a message passing implementation matching the MPJ specification is obvious. The MPJ implementation should be flexible and efficient. Portability is being guaranteed by Java, but not flexibility and efficiency.

RMI and Java Sockets usually use TCP for network communication. That makes them not flexible enough for high performance computing, for example on a Myrinet computer cluster. RMI, as shown in Section 3.1.2, is not efficient enough to satisfy the requirements of a message passing platform. However, Ibis provides solutions

[1] The Java Grande Forum online at http://www.javagrande.org

to both efficiency and flexibility, and thus offers an excellent foundation to build an MPJ implementation on top of it. In the following this implementation is called MPJ/Ibis.

In contrast to MPI, MPJ does not address the issue of thread safety explicitly. Synchronizing multiple threads in asynchronous communication models - message passing models express it by the non-blocking communication paradigm - is not trivial. Since synchronizing is still an expensive operation [8] even in cases when it is not used but implemented, MPJ/Ibis avoids the overhead of being thread safe. Multithreaded applications on top of MPJ must synchronize the entry points to MPJ/Ibis, that means only one thread is allowed to call MPJ primitives at a time. Thus, it will be ensured that the requirement of getting the most efficient result is satisfied in this aspect.

4.2 MPJ Specification

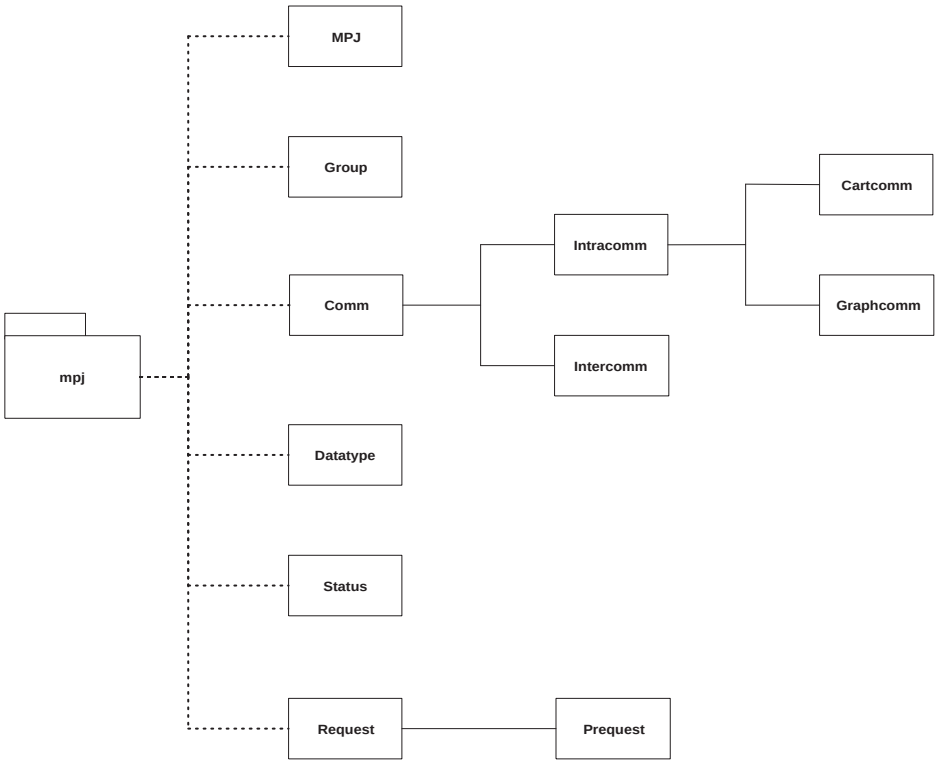

Figure 4.1: Principal Classes of MPJ (after [4])

MPJ is a result from the adaption of the C++ MPI bindings specified in MPI-2 [16] to Java. The class specifications are directly built on the MPI infrastructure provided by MPI-1.1 [15]. It has been announced that the extensions of MPI-2 like dynamic process management will be added in later work, but that has not been done yet. To stay conform to the specification, the extensions are unsupported by MPJ/Ibis as well. Figure 4.1 shows the most important classes of MPJ, which will be briefly introduced in the following.

All MPJ classes are organized in the package *mpj*. The class *MPJ* is responsible for initialisation of the whole environment, like global constants, peer connections and the default communicator *COMM_WORLD*. Thus, all members of *MPJ* are defined static. As explained in Section 2.5, processes of one group, represented by the class *Group*, communicate via the communicators. All communication elements are instances of class *Comm* or its subclasses. For example *COMM_WORLD* is an instance of class *Intracomm*.

The point-to-point communication operations, such as *send*, *recv*, *isend* and *irecv*, are in the *Comm* class. For example, the method prototype of *send* looks like this:

```
void Comm.send(Object buf, int offset, int count, Datatype datatype,
               int dest, int tag) throws MPJException
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	data type of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Table 4.1: MPJ *send* prototype (after [4])

The message to be sent must consist either of an array of primitive types or of an array of objects, which has to be passed as an argument called `buf`. Objects need to implement the *java.io.Serializable* interface. The `offset` indicates the beginning of the message (see Table 4.1).

The `datatype` argument is necessary to support derived datatypes in analogy

to MPI (see Section 2.2). These `datatypes` are instances of the class *Datatype* and must match the base type used in `buf`. MPJ specifies the basic datatypes. Table 4.2 shows a list of the most important predefined types.

MPJ.BYTE	MPJ.CHAR
MPJ.BOOLEAN	MPJ.SHORT
MPJ.INT	MPJ.LONG
MPJ.FLOAT	MPJ.DOUBLE
MPJ.OBJECT	...

Table 4.2: MPJ basic datatypes (after [4])

As shown in the example above the *send* operation does not have a return value, since it is a blocking operation. In contrast to MPI all communication operations use Java's exception handling to report errors. Additionally non-blocking operations, such as *isend* or *irecv*, always return a *Request* object. Requests represent ongoing message transfers and provide several methods to obtain knowledge about the current state. Once a message is completed, those methods return a *Status* object, which contains detailed information about the message transfer. *Prequest*, an extension to *Request*, is the result of a prepared persistent message transfer. Persistent communication is organized in class *Comm* and will not discussed here, because it is of minor importance, even it has been implemented in MPJ/Ibis. The collective communication operations are part of the class *Intracomm*, which extends *Comm*, so that it is possible for them to use the point-to-point primitives.

4.3 MPJ on Top of Ibis

Ibis/MPJ has been divided into three layers and was built directly on top of the IPL (see Figure 4.2).

The *Ibis Communication Layer* provides the low level communication operations. The *Base Communication Layer* takes care of the basic send and receive operations specified by MPJ. It includes the blocking and nonblocking operations and the various test and wait statements.

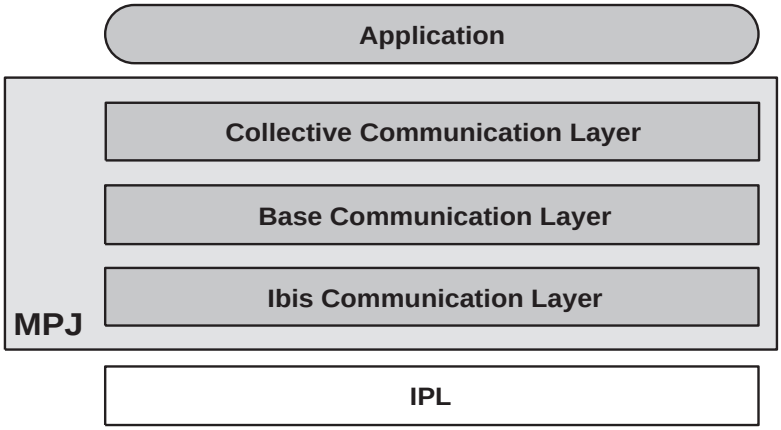

Figure 4.2: MPJ/Ibis design

It is also responsible for group and communicator management. The *Collective Communication Layer* implements the collective operations on top of the *Base Communication Layer*. An MPJ/Ibis application is able to access both the *Base* and the *Collective Communication Layer*.

4.3.1 Point-to-point Communication

Each participating process is connected explicitly to each other using the IPL's send and receive ports. Ibis offers two mechanisms for data reception, upcalls and downcalls (see Section 3.2). Since upcalls always require an additional running thread collecting the messages, the performance of MPJ/Ibis would be affected negatively in all cases while switching between running threads. However, threads can not be avoided completely, but the amount should be reduced to a minimum. Therefore, MPJ/Ibis has been designed to use downcalls. A pair of a send and a receive port to one communication partner is summarized in a *Connection* object and all existing *Connections* are collected within a lookup Table called *ConnectionTable*.

The IPL does not provide primitives for non-blocking communication. To support blocking and non-blocking communication the *Ibis Communication Layer* has been designed to be thread safe. That allows non-blocking communication on top of the blocking communication primitives using Java threads. Multithreading cannot be avoided in this case. Furthermore, Ibis only provides communication in eager send mode, while handshaking is not supported. While MPIs and MPJs ready and

synchronous send modes require some kind of handshaking in order for a better buffer organization for short and large messages, the IPL implementations take care about that automatically. Therefore MPJ/Ibis does not differ between ready and synchronous send modes, but buffered send is supported.

Message Send Internally messages are represented by the *MPJObject* class, which consists of a header and a data part. The header is an one-dimensional integer array containing the message envelope values. Sending a message has been implemented in a straight forward way, as shown in Figure 4.3. Once one of the send primitives of the *Base Communication Layer* has been called, the assigned thread checks if another send operation is in progress. In that case, it has to wait until the previous operation has been finished. Waiting, lock obtaining and releasing have been left to Java's synchronization mechanisms. Instead of writing the *MPJObject* directly to the send port, which causes unnecessary object serialization, the send operation has been divided into three steps. First the header will be written explicitly. Second, it is determined if the message data has to be serialized into a system buffer or not before it can be sent. Third, depending on step two the message data or the system buffer will be written to the send port.

Message receive Since non-blocking communication allows to vary the order of calling receive primitives, it is necessary to add a queue for every receive port, where unexpected messages will be collected. Usually, receiving messages into a queue will be done by using a permanent receiver thread, which is connected to the receive port and manages the queue filling automatically. Therefore, one thread would be needed for each receive port, resulting in a slowdown of the whole system's performance, due to thread switching of multiple threads depending on the number of participating processes. Another disadvantage of such a concept is, that zero-copying will not be possible, since all messages have to be copied out of the queue.

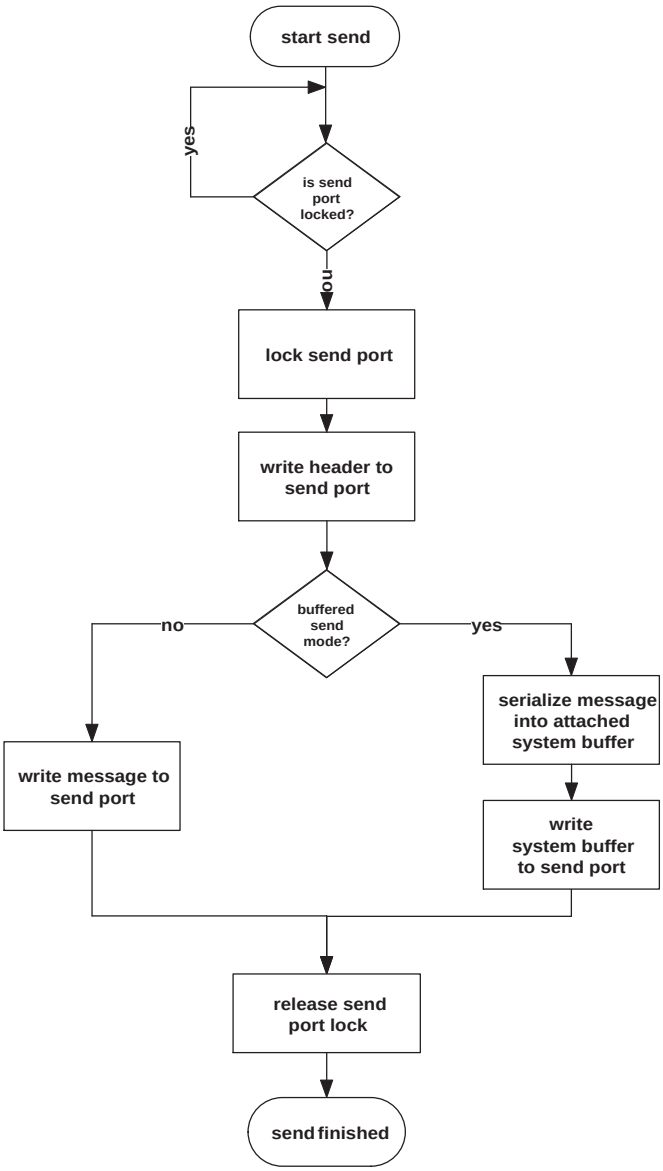

Figure 4.3: MPJ/Ibis send protocol

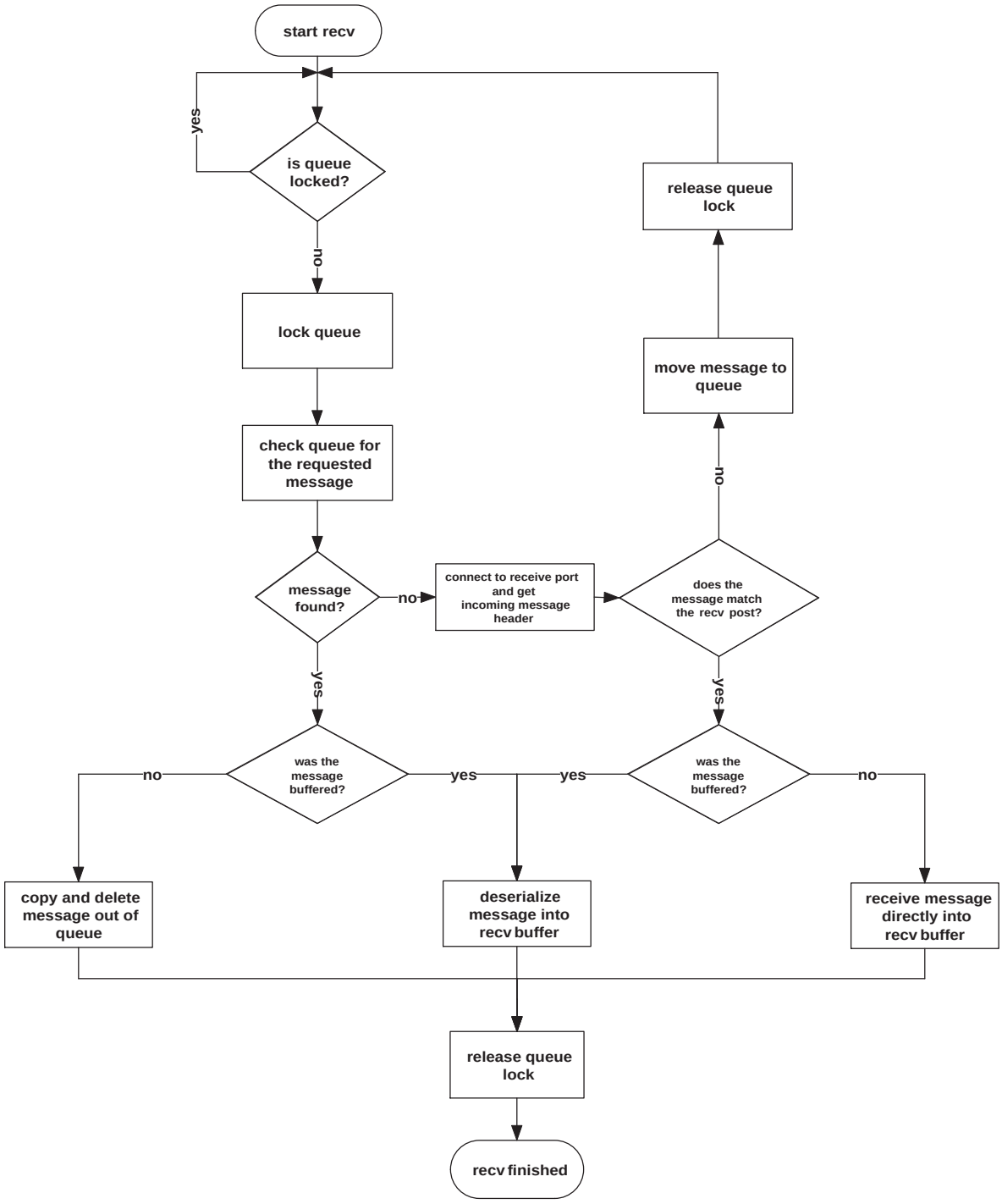

Figure 4.4: MPJ/Ibis receive protocol

To avoid continuous thread switching and to allow zero-copying the receive protocol has been designed in way that allows a receiving primitive to connect explicitly to the receive port. The receive protocol is shown in Figure 4.4. After obtaining the lock, a receiver thread checks whether the queue contains the message requested. If the message has been found, it will be copied or deserialized out of the queue, depending whether the message was buffered or not. If the message was not found inside the queue, the receiver connects to the receive port and gets the incoming message header to determine, if the incoming message is targeted to the receiver. If not, the whole message including the header will be inserted into an *MPJObject* and moved into the queue where it waits for the matching receiver. To give other receiving threads the chance to check the queue, the lock will be temporarily released. If the message header from the receive port matches the posted receive, the non-buffered message will be received directly into the receive buffer, while a buffered message has to be deserialized into the receive buffer.

4.3.2 Groups and Communicators

As mentioned in Section 2.5, each group of processes has a communicator that is responsible for message transfers within the group. Since communicators can share the same send and receive ports, it is mandatory to prevent mixing up messages of different communicators. The *tag* value, which is used by the user to distinguish different messages, is not useful for this case. Therefore, on creation each communicator gets a unique *contextId*, which allows the system to handle communication of different communicators on the same ports at the same time. When sending a message, the message header will be extended by adding the *contextId* of the used communicator to it. Messages are identified by both the *tag* value and the *contextId*.

Creating a new communicator is always a collective operation. Each process holds the value of the highest *contextId* that is used locally. When a new communicator is going to be created, each process creates a new temporary *contextId* by increasing the highest by one. To ensure that all processes use the same *contextId* for the new communicator, the temporary *contextId* will be allreduced to the maximum. After that, the new communicator will be created and the local system

informed that the new *contextId* is the highest.

4.4 Collective Communication Algorithms

The collective operations of class *Intracomm* have been implemented on top of the point-to-point primitives. Since the naive approach of sending messages directly may result in high latencies, those operations should use specialized algorithms. For example, letting the root in a *broadcast* operation send a message to each node explicitly is inefficient, while the last receiver has to wait until the message has been sent to the other processes. The current MPI implementations contain a vast amount of different algorithms realizing the collective operations and research to their optimization is not finished yet.

Collective Operation	Algorithm	Upper Complexity Borders
allgather	double ring	$O(n)$
allgatherv	single ring	$O(n)$
allreduce	recursive doubling	$O((log\, n) + 2)$
alltoall	flat tree	$O(n^2)$
alltoallv	flat tree	$O(n^2)$
barrier	flat tree	$O(2n)$
broadcast	binomial tree	$O(log\, n)$
gather	flat tree	$O(n)$
gatherv	flat tree	$O(n)$
reduce	commutative op: binomial tree non-commutative op: flat tree	$O(log\, n)$ $O(n)$
reduceScatter	phase 1: reduce phase 2: scatterv	commutative op: $O((log\, n) + n)$ non-commutative op: $O(2n)$
scan	flat tree	$O(n)$
scatter	flat tree	$O(n)$
scatterv	flat tree	$O(n)$

Table 4.3: Algorithms used in MPJ/Ibis to implement the collective operations

MPJ/Ibis provides a basic set of collective algorithms, which may be extended and more optimized in future work. At least one for each operation has been implemented. Table 4.3 shows the algorithms used for all collective operations including their upper complexity borders in O-notation, where n is the number of the pro-

cesses involved. In accordance to the MPI specification four collective operations have been extended to achieve more flexibility. The extended operations are called *allgatherv, alltoallv, gatherv* and *scatterv*. They allow to vary the item sizes and buffer displacements explicitly for each process. In the following the algorithms used will be introduced exemplary.

Flat Tree The flat tree model follows the naive approach mentioned above. The root process P_0 sends to and/or receives from the other group members directly. Figure 4.5 demonstrates the scatter operation using the flat tree communication scheme with five participating processes. In each step the root sends a message containing the elements to be scattered to the next process. The number of steps needed depends linearly on the number of processes. Figure 4.6 shows a different view of the same scatter scheme, which results in a tree with n-1 leaves and exactly one parent node, namely the root.

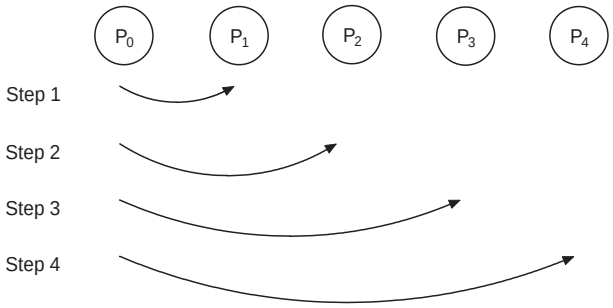

Figure 4.5: *Scatter* sending scheme

Typically the other operations using the flat tree model have a complexity of $O(n)$ as well, except *alltoall* and *barrier*. The *alltoall* operation has been implemented using the flat tree, but each process creates a flat tree to send messages to each other. This results in n flat trees with an overall upper complexity border of $O(n^2)$.

The *barrier* operation uses the process with rank zero to gather zero sized messages from the other processes. These messages will be scattered back to complete the operation. Two flat trees are needed with a total cost of $O(2n)$.

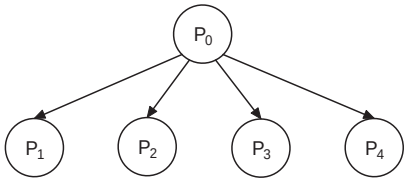

Figure 4.6: Flat tree view

Binomial Tree The *broadcast* follows the binomial tree model shown in Figure 4.8. Sending messages via the binomial tree structure has a complexity of $O(log\, n)$. In Figure 4.7 the *broadcast* operation takes place with eight participating processes, where P_0 is the root sending its send buffer to the other processes. After each step the number of sending processes is doubled. In this example three steps are needed to execute the whole operation, while a *broadcast* in flat tree model would take seven steps.

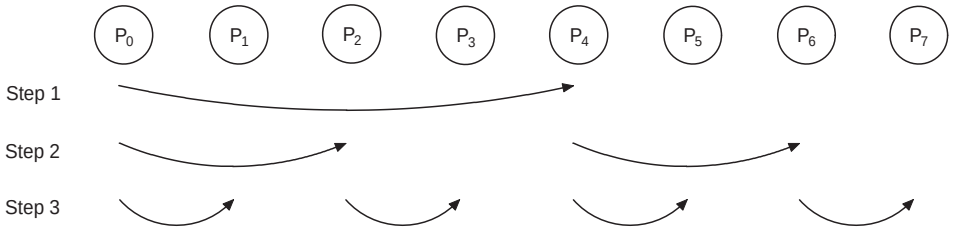

Figure 4.7: *Broadcast* sending scheme

The *reduce* operation has been implemented using a binomial tree in reverse order. After receiving a message each process combines its send and receive buffer using the assigned operation. The result will be written into the send buffer and represents the argument for the next step. In the end the reduce result appears at the root process. Since user-defined operations may be non-commutative and the binomial tree model does not follow strict ordering, this algorithm is not universally valid. In the case of non-commutative user-defined operations the flat tree model will be used instead.

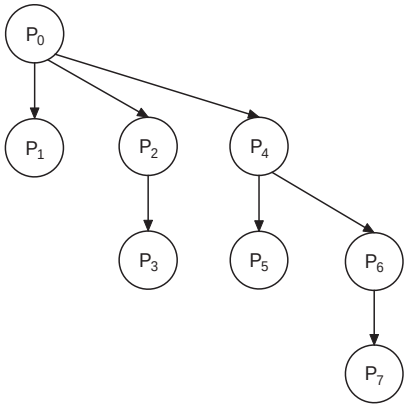

Figure 4.8: Binomial tree view

Ring In *allgatherv*, each process first sends its item to gather to its right neighbour (the process with the next higher rank). If the rank of a process is $n-1$ then it sends its item to the process with rank 0. Second, in the next steps each process sends the received item to its right neighbour. The execution is complete after $n-1$ steps, when each process has received the items of all the other processes. Figure 4.9 illustrates one step of the ring algorithm.

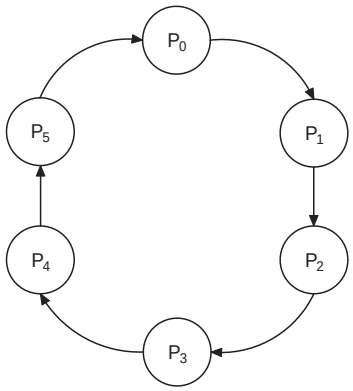

Figure 4.9: Ring sending scheme (only 1 step)

Allgather uses an extension to the ring model mentioned above, called double ring. With double ring all processes send their items to gather to their right and left neighbours. Therefore the execution completes after $(n-1)/2$ steps, but in each step the number of send and receive operations needed is doubled compared to the ring model. Overall this model shows the same complexity of $O(n)$ as the ring.

Recursive Doubling In Recursive doubling, used by allreduce, in the first step all the processes which have a distance of 1 exchange their messages followed by a local reduction (see Figure 4.10). In the next steps the distances will be doubled each time. In the end, after $log\, n$, steps the allreduce operation has been finished for the case that the number of participating processes is a power-of-two. For the non-power-of-two case the number of processes performing the recursive doubling will be reduced to power-of-two. The remaining processes send their items to the processes of the recursive-doubling-group explicitly before the doubling starts. When the recursive doubling has finished, the reduced values will be sent to the remaining processes. That causes two extra steps in the non-power-two case and results in a complexity of $O((log\, n) + 2)$.

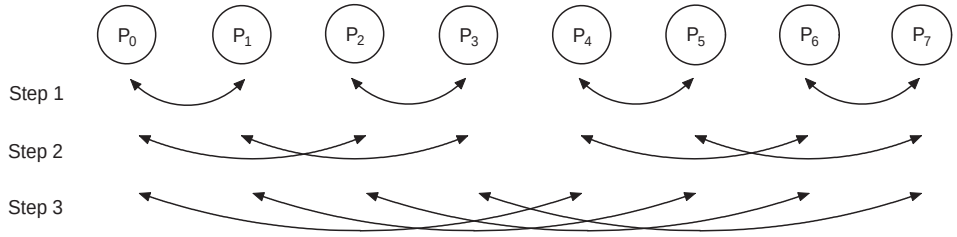

Figure 4.10: Recursive doubling illustration

4.5 Open Issues

Multidimensional Arrays & Derived Datatypes Since the MPJ specification only specifies generic send and receive primitives, which expect message data of the type *java.lang.Object*, MPJ/Ibis has to cast explicitly the real type and dimension of the arrays used. That makes it impossible for MPJ/Ibis to support all dimensions of arrays. In contrast to the programming language C and others, for which the original MPI standard was presented, Java represents multidimensional arrays as arrays of arrays. Traversing those arrays in Java with only one pointer is impossible. Therefore MPJ/Ibis supports only one-dimensional arrays. Multidimensional arrays can be sent as an object or each row has to be sent explicitly.

Since Java provides derived datatypes natively using Java objects there is no real need to implement derived datatypes in MPJ/Ibis. Nevertheless contiguous

derived datatypes are supported by MPJ/Ibis to achieve the functionality of the reduce operations MINLOC and MAXLOC specified by MPJ, which need at least a pair of values inside a one-dimensional array. The other types of derived datatypes may be implemented in future work, if multidimensional arrays will be supported directly.

Other Issues Due to time constraints for this thesis, MPJ/Ibis supports creating and splitting of new communicators, but intercommunication is not implemented yet (see Section 2.5). At this moment, MPJ/Ibis also does not support virtual topologies (see Section 2.6). Both may be added in future work as well.

Chapter 5

Evaluation

5.1 Evaluation Settings

MPJ/Ibis on top of Ibis Version 1.1 has been evaluated on the Distributed ASCI Supercomputer 2 (DAS-2) with 72 nodes in Amsterdam. Each node consists of:

- Two 1-Ghz Pentium-IIIs
- 1 GB RAM
- a 20 GByte local IDE disk
- a Myrinet interface card
- a Fast Ethernet interface (on-board)

The operating system is Red Hat Enterprise Linux with kernel 2.4. Only one processor per node has been used during the evaluation.

To achieve more comparable results the following benchmarks have been performed with mpiJava [1] as well. MpiJava is based on wrapping native methods like the MPI implementation MPICH with the Java Native Interface (JNI). Here, mpiJava Version 1.2.5 has been bound to MPICH/GM Version 1.2.6 for Myrinet. For Fast Ethernet there was only MPICH/P4 available on the DAS-2, which is not compatible to mpiJava. Nevertheless, the values for MPJ/Ibis on TCP for Fast Ethernet will be presented as well. Both MPJ/Ibis and mpiJava have been evaluated using Suns JVM Version 1.4.2.

For Ibis two different modules exist to access a Myrinet network, called Net.gm and Panda. During the evaluation it has been found out, that Net.gm in some cases causes deadlocks, due to problems in buffer reservation, when multiple objects are being transfered from multiple senders to one recipient. The Panda implementation has shown memory leaks for large message sizes resulting in performance reduction and deadlocks. However, where it is possible the results for MPJ/Ibis on Myrinet will be presented for each benchmark. MPJ/Ibis on TCP performed stable.

MpiJava on MPICH/GM in some cases performed unstable resulting in memory overflows and broken data streams. These misbehaviours occurred randomly and could not be reproduced.

5.2 Micro Benchmarks

The micro benchmarks, which have been implemented for Ibis, MPJ/Ibis and mpiJava, firstly measure the round trip latency by sending one byte back and forth. Since the size can be neglected, the round trip latency is divided by two to get the latency for a one way data transfer. Secondly, the micro benchmarks measure the execution time of sending an array of doubles from one node to a second, which acknowledges the reception by sending one byte back to the sender. This is repeated for array sizes from 1 byte to 1MB. Thirdly, the throughput of object arrays is measured, where each object contains a byte value. The measurement is repeated for different array sizes in analogy to the second step.

Implementation	latency [μs]
mpiJava (MPICH/GM)	28
Ibis (Panda)	44
Ibis (Net.gm)	52
MPJ/Ibis (Panda)	50
MPJ/Ibis (Net.gm)	53
Ibis (TCP)	113
MPJ/Ibis (TCP)	120

Table 5.1: Latency benchmark results

Latencies Table 5.1 shows the latency benchmark results for MPJ/Ibis, Ibis and mpiJava. On Myrinet, MPJ/Ibis and Ibis have considerably higher latencies than mpiJava. The reason of the gap between Ibis and mpiJava is beyond the objective of this thesis. Furthermore, MPJ on top of Ibis does not show considerably higher latencies than Ibis itself. Thus, the message creation overhead of MPJ/Ibis does not influence the latency by a large margin.

Throughput Double Arrays The figures 5.1 and 5.2 show the throughput measurement results for Ibis and the message passing implementations. For TCP Ibis and MPJ/Ibis almost use the whole available bandwidth provided by Fast Ethernet for data sizes greater than 1KB. For sizes beyond 32KB the performance of the Panda implementation breaks down, caused by the memory leaks mentioned above. This halves the throughput for Ibis and MPJ/Ibis.

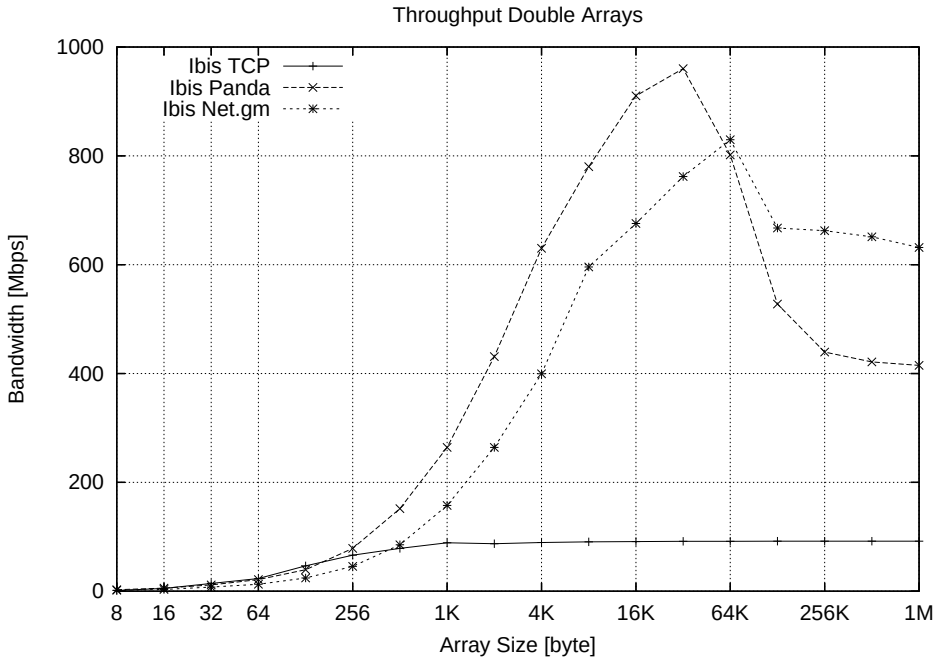

Figure 5.1: Double array throughput in Ibis

Net.gm does not reach the maximum of Panda, but for large data sizes (> 64KB) it is much faster. MPJ/Ibis on top of Net.gm also outperforms mpiJava. The break down of mpiJavas performance is caused by MPICH/GM switching from ready send mode to synchronous send mode at message sizes beyond 128KB. All

message passing implementations do not take advantage of the available bandwidth provided by Myrinet.

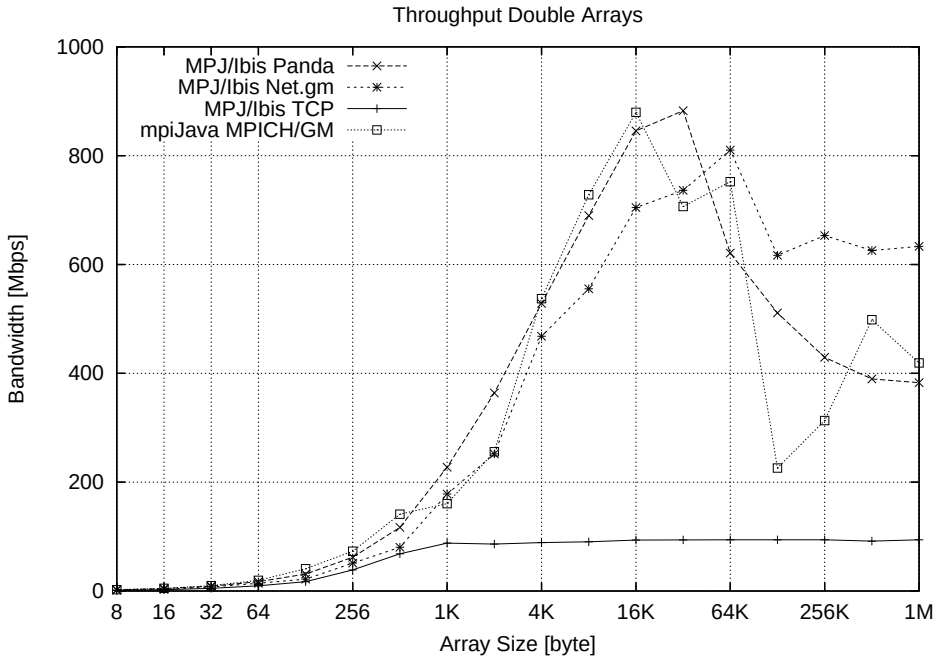

Figure 5.2: Double array throughput in MPJ/Ibis and mpiJava

Throughput Object Arrays The throughput of object arrays is not limited by the physical restrictions of the underlying network hardware. Objects have to be serialized at the senders side and deserialized by the receiver. The performance gap between Ibis and MPJ/Ibis is small (compare Figures 5.3 and 5.4). Both use the Ibis serialization model introduced in Section 3.3, which still is a performance limiter. On the other hand MPJ/Ibis outperforms mpiJava by a large margin, which depends on the serialization model provided by Sun's JVM. MpiJava does not even reach 15 Mbps on Myrinet when object arrays are being transfered.

Each introduced serialization model has to perform a duplicate detection before sending an object, in a way that no object needs to be transfered twice. Therefore, each object reference has to be stored in a hash table to allow a lookup if an object to be sent has been processed before. For larger arrays the hashtable becomes larger as well causing communication slow downs. These slow downs have been shown by all implementations, when the object array sizes grow beyond 8KB.

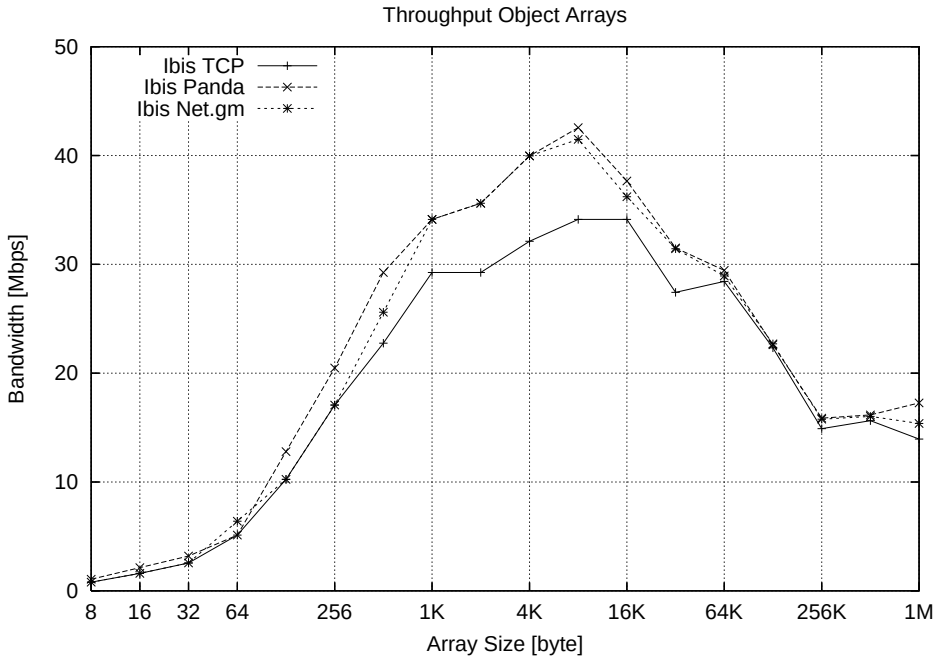

Figure 5.3: Object array throughput in Ibis

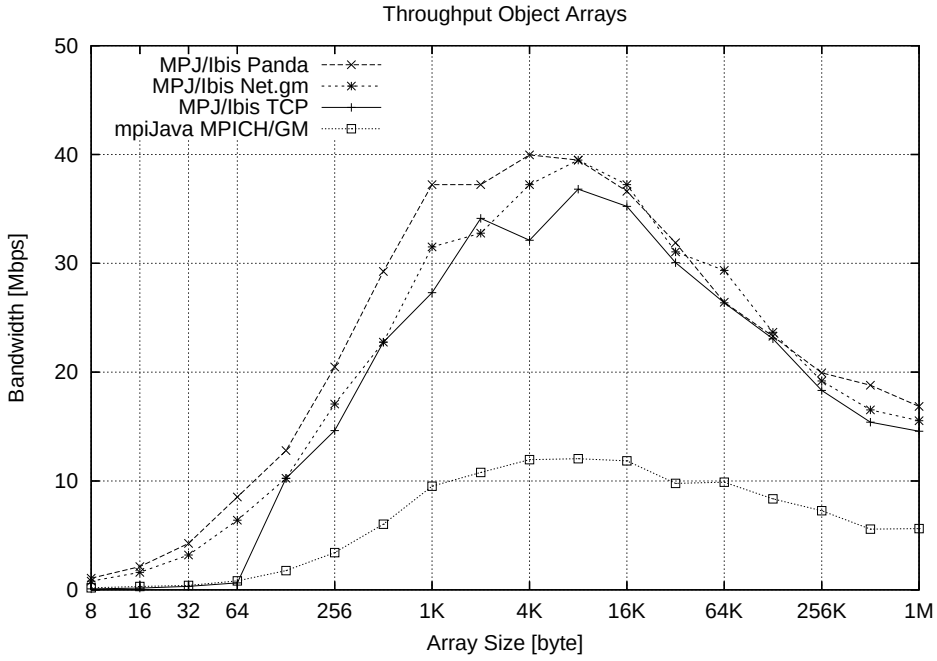

Figure 5.4: Object array throughput in MPJ/Ibis and mpiJava

5.3 Java Grande Benchmark Suite

The Java Grande Benchmark Suite[1] maintained by the Edinburgh Parallel Computing Centre (EPCC) consists of three Sections. Section 1 performs low-level operations, pingpong for throughput measurements and several benchmarks for the collective operations. Section 2 provides five kernel applications performing common operations, that are widely used in high performance computation. In section 3 three large applications are benchmarked. The benchmarks of section 2 and 3 measure execution times. Those results will be presented in relative speedup [10, 30], which is defined as follows:

$$relative\,speedup\,(p\ \ processors) = \frac{runtime\,par.\,alg.\,(1\ \ processor)}{runtime\,par.\,alg.\,(p\ \ processors)}$$

Additionally the theoretical perfect speedup for the kernel and application benchmarks is marked in each presentation.

Originally the benchmark suite has been implemented to match mpiJava's API, which is slightly different to that of MPJ. Thus, the benchmark suite has been ported to MPJ. The sections 2 and 3 contain predefined problem sizes to be solved, which were not large enough to perform efficiently on the DAS-2, when computation time becomes short. Where possible the predefined problem sizes have been increased to improve the use of capacity provided by the existing hardware.

5.3.1 Section 1: Low-Level Benchmarks

The low level benchmarks are designed to run for a fixed period of time. The number of operations executed in that time is recorded, and the performance reported as operations/second for the barrier and bytes/second for the other operations. Both the size and type of arrays transfered are varied. The type is either a double, or a simple object containing a double, which allows to compare the communication overhead of sending objects and basic types.

[1] Java Grande Benchmark Suite online at http://www.epcc.ed.ac.uk/javagrande/mpj/contents.html

Pingpong The pingpong benchmark measures the bandwidth achieved to send an array back and forth between exactly two nodes. The aggregated results of the double array and the object array benchmarks are shown in the Figures 5.5 and 5.6.

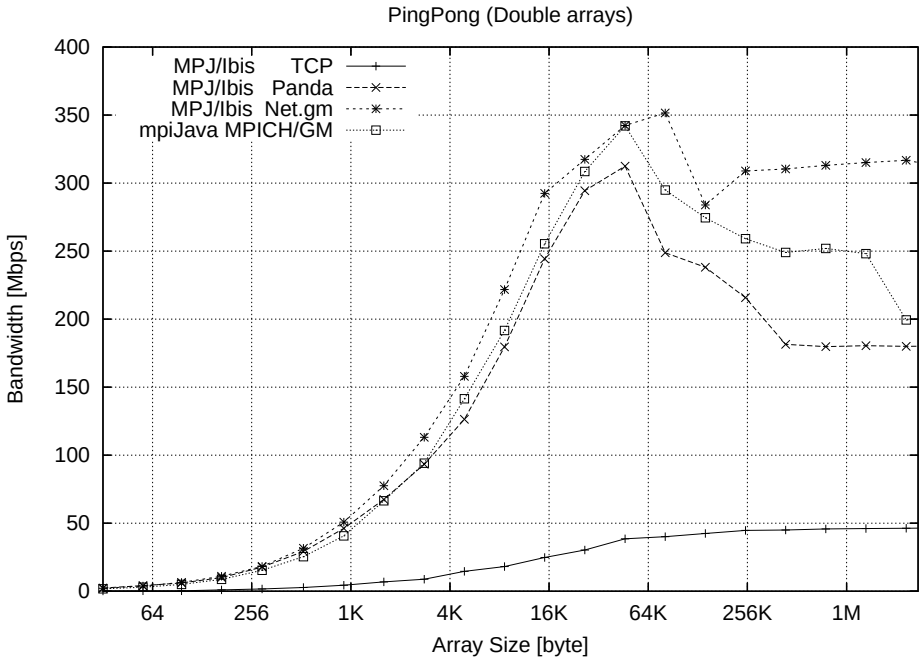

Figure 5.5: Pingpong benchmark: arrays of doubles

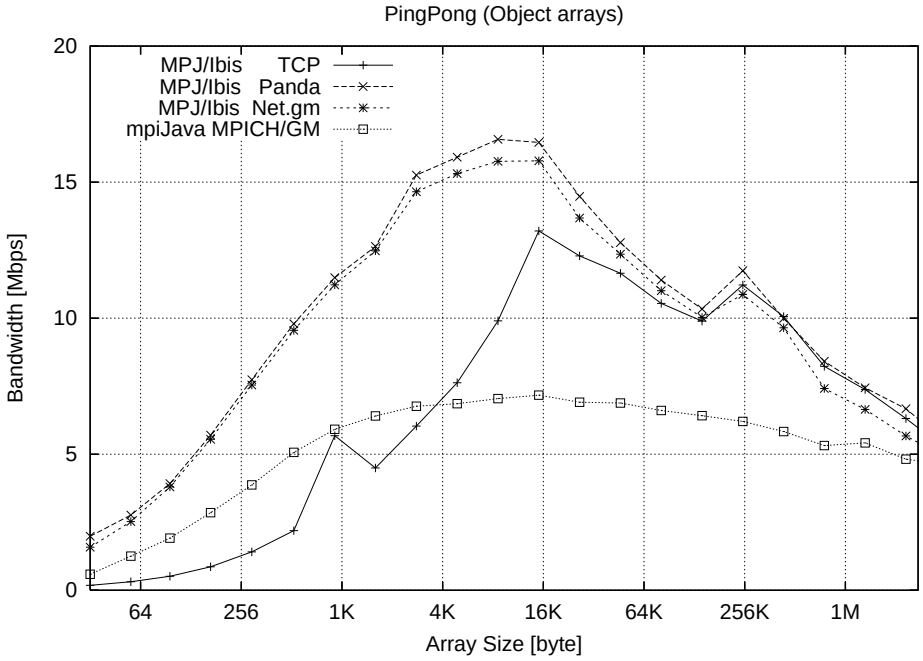

Figure 5.6: Pingpong benchmark: arrays of objects

As can be seen in Figure 5.5 for all Myrinet implementations the achieved throughput rises in the same way up to array sizes of almost 64KB. Beyond 64KB MPJ/Ibis on Net.gm keeps outperforming mpiJava, while MPJ/Ibis on Panda slows down due to the mentioned memory leaks. For object arrays mpiJava uses the serialization mechanism provided by the JVM, while MPJ/Ibis takes advantage of the Ibis serialization. That results in higher throughputs, even at array sizes greater than 4KB MPJ/Ibis on TCP performs better than mpiJava. With arrays larger than 1MB the performance advantage of the Ibis serialization almost disappears for Ibis' Myrinet implementations. Overall these results reflect the measurements of the micro benchmarks (see Section 5.2).

Barrier MPJ/Ibis' implementation of the *barrier* operation is not optimal (see Figure 5.7). MPICH/GM uses the recursive doubling algorithm, which has a lower execution time than the algorithm used in MPJ/Ibis. In both MPJ/Ibis and mpiJava a zero sized byte array is being transfered in each communication step. Additionally, due to higher zero-latencies the relative performance of MPJ/Ibis compared to mpiJava is further slowed down.

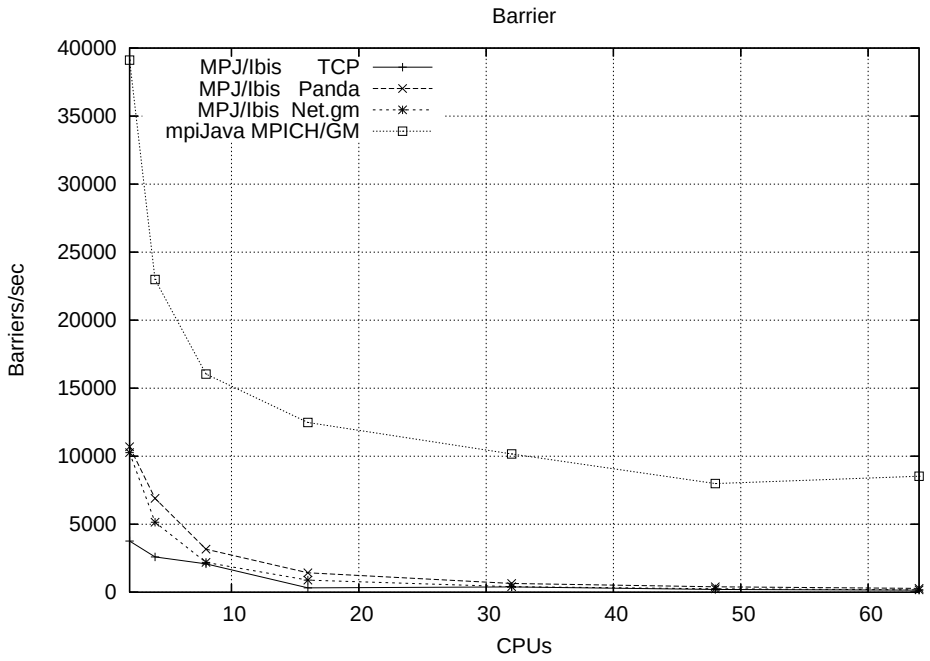

Figure 5.7: Barrier benchmark

Broadcast The *broadcast* benchmark performs in a similar way as the pingpong benchmark. In addition to pingpong this benchmark is not restricted to only two nodes and therefore the broadcast operation has been evaluated on up to 48 nodes. MPJ/Ibis and MPICH/GM implement the *broadcast* operation using the same algorithm.

Figure 5.8 shows the results of MPJ/Ibis and mpiJava. For double arrays mpiJava performs better than MPJ/Ibis up to eight involved nodes, caused by higher throughputs. With more participating processes the difference between the implementations working on Myrinet becomes marginally small.

Broadcasting arrays of objects comes with a considerably performance advantage to MPJ/Ibis. On two processors MPJ/Ibis outperforms mpiJava almost by a factor of about six. By increasing the number of nodes to 48 processes this gap rises to a factor of about 20, caused by the more efficient Ibis serialization. Overall the results of the *broadcast* benchmark correspond to those of the micro and pingpong benchmarks.

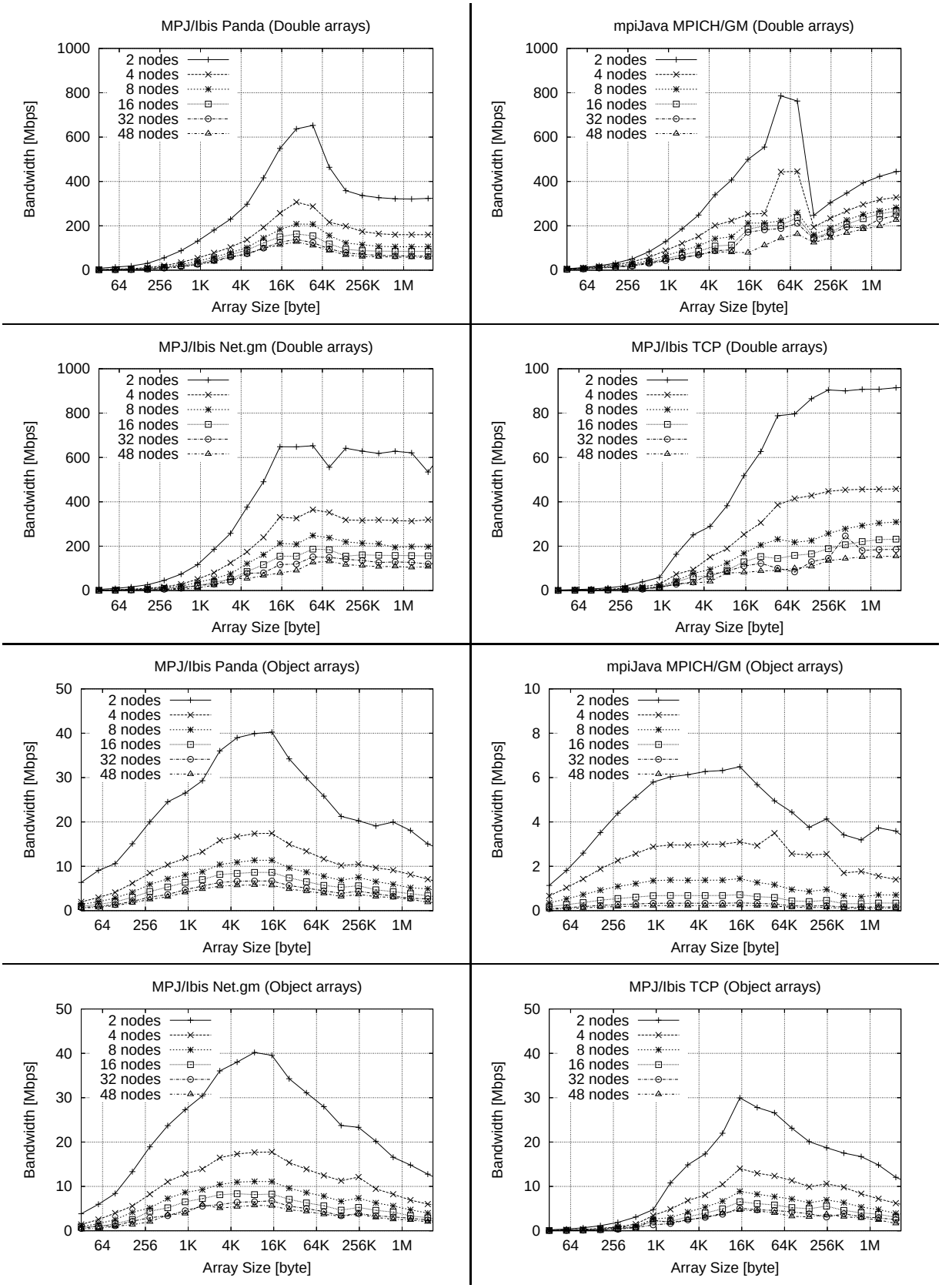

Figure 5.8: Broadcast benchmark

Reduce The *reduce* benchmark only uses double arrays, since the built in reduce operations do not support object arrays. Here, the arrays will be reduced by adding the array items using the sum operation.

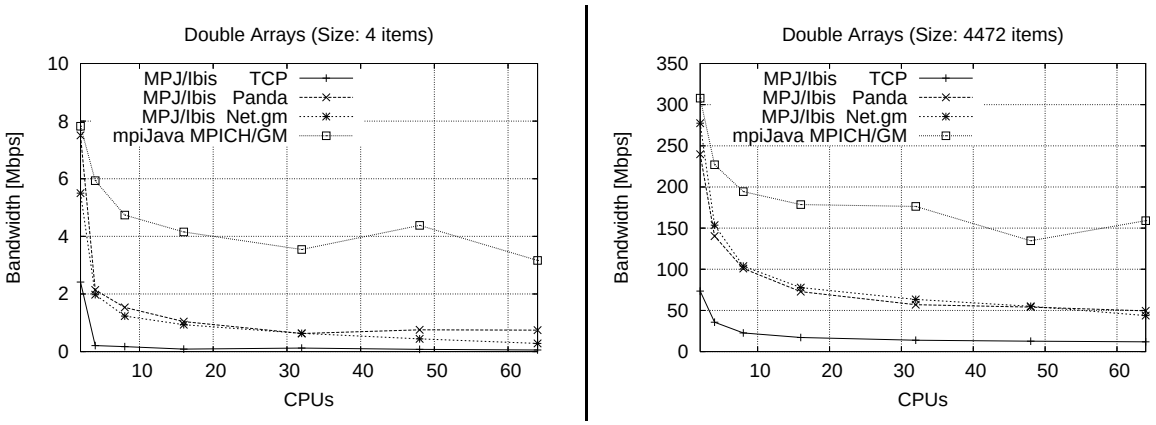

Figure 5.9: Reduce benchmark

In general mpiJava shows a better performance (see Figure 5.9), although MPICH/GM implements the same tree algorithm as MPJ/Ibis. In both, before executing the *reduce* operation a temporary buffer for array reception will be created at each node. Since Java fills arrays with zeros at initialization in contrast to C, the overhead for MPJ/Ibis is much higher. Furthermore the smaller throughput results for MPJ/Ibis are also indicated by the higher latencies shown by the micro benchmarks.

Scatter Scattering messages in mpiJava and MPJ/Ibis has been implemented in the same way. As for the *reduce* operation the *scatter* benchmark only measures the throughput for two different sizes, but for double and object arrays. It was not possible to run this benchmark on the Net.gm implementation for MPJ/Ibis, which caused deadlocks when more than two processes were involved. As can be seen in Figure 5.10 for small double arrays mpiJava shows less performance loss on larger numbers of nodes than MPJ/Ibis on Panda, because of the lower latency shown in section 5.2. For larger double arrays the impact of the latency gap becomes marginally small. As expected for object arrays MPJ/Ibis on Panda outperforms mpiJava almost by a factor of 1,9. Overall the performance of scattering messages is highly influenced by the flat tree algorithm used.

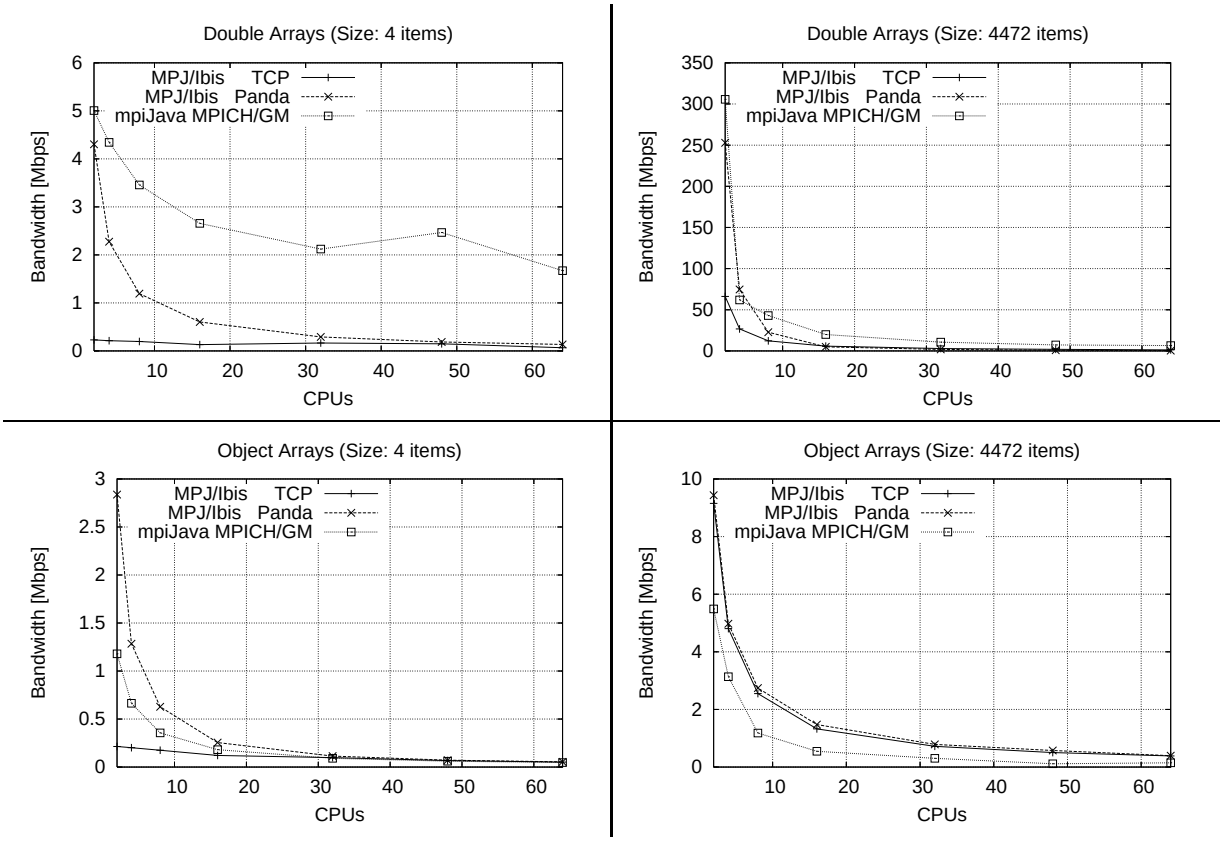

Figure 5.10: Scatter benchmark

Gather It was not possible to run the *gather* benchmark neither with mpiJava nor with the Myrinet implementations of Ibis. With all of them this benchmark exceeded the limit of the physical memory provided by each node of the DAS-2. Only MPJ/Ibis on top of TCP worked stable. The results are presented in Figure 5.11.

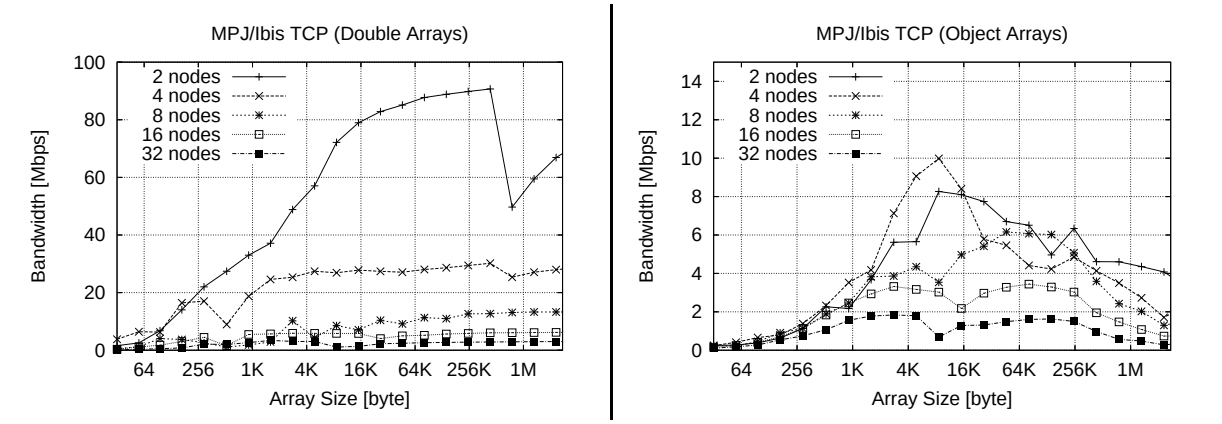

Figure 5.11: Gather benchmark

This benchmark functions the same as the *broadcast* benchmark. For double arrays the *gather* operation shows higher performance loss on more than two nodes than *broadcast*. This is because of the flat tree algorithm, in which each node sending a message to the root has to wait until the root has received the message from the previous node. On a growing number of processes this leads to substantially higher latencies.

For object arrays the results show an unstable behaviour of MPJ/Ibis on TCP using the *gather* operation up to eight processes involved. While MPJ/Ibis for double arrays works as expected (more processes causing less throughput), the results for object arrays lead to the assumption that Ibis serialization for local copies may cause inefficiencies. In each call of *gather* the root has to copy the items of its send buffer locally into the receive buffer. Object arrays will be copied using Ibis serialization. Particularly the relative part of the local copy to the communication overhead at a smaller number of processes is much higher than at larger numbers of processes. This impact should be elaborated more in future work.

Alltoall As with the *gather* benchmark it was not possible to perform the *alltoall* benchmark with mpiJava and MPJ/Ibis on top of Panda and Net.gm. With all Myrinet implementations this benchmark produced memory overflows. Additionally mpiJava randomly reported broken data streams.

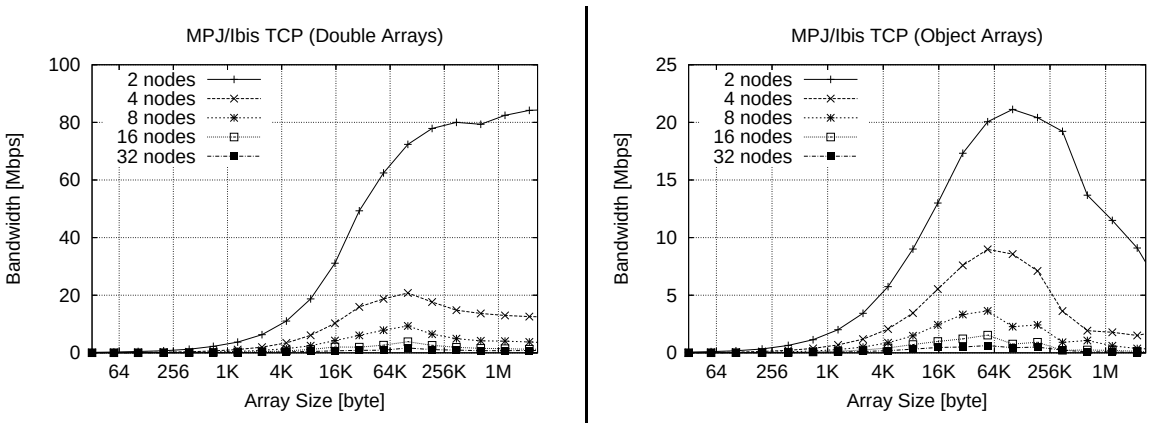

Figure 5.12: Alltoall benchmark

In MPJ/Ibis the *alltoall* operation uses non-blocking communication primitives,

which are called simultaneously. On TCP multiple threads competing for system resources do notably interfere the communication performance leading to high performance losses the more processes are involved.

5.3.2 Section 2: Kernels

Crypt Crypt performs an IDEA (International Data Encryption Algorithm) encryption and decryption on a byte array with a size of $5 * 10^7$ items. Node 0 creates the array and sends it to the other nodes, where the encryption and decryption takes place. After computation the involved nodes send their results back to node 0 using individual messages. The time measurement starts after sending the initialized arrays and stops when node 0 has received the last array.

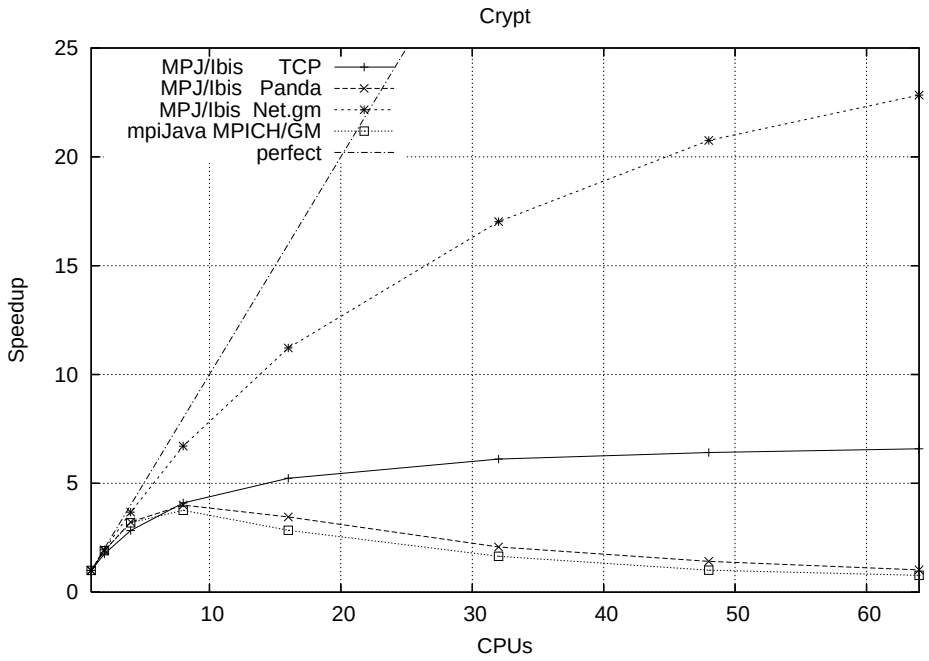

Figure 5.13: Crypt speedups

The crypt kernel does not scale perfectly in all cases (see Figure 5.13). However, MPJ/Ibis on top of Panda and mpiJava show the same speedups. Beyond 8 nodes both break down. As expected from the micro benchmarks the impact of the communication overhead of Panda becomes less neglectable on a growing number of nodes involved, due to reduced computation time. MPJ/Ibis on Net.gm shows

the highest speedup of up to about 23 on 64 CPUs and outperforms mpiJava by a large margin. Even MPJ/Ibis on TCP shows a better performance than mpiJava on Myrinet.

LU Factorization This kernel solves a linear system containing 6000 x 6000 items using LU factorization followed by a triangular solve. The processes exchange double and integer arrays using the broadcast operation. The time during factorization including communication is measured.

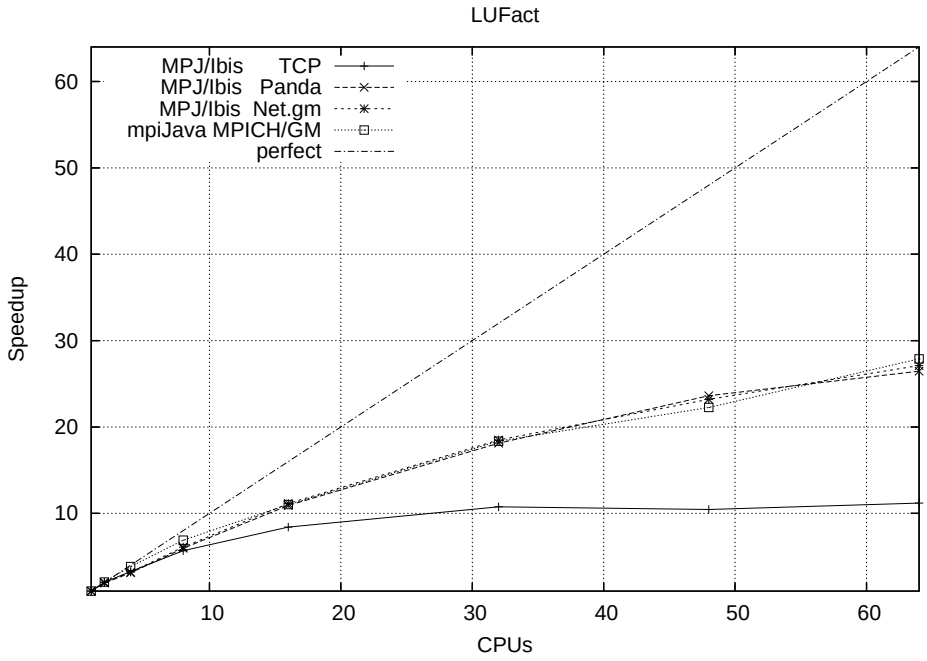

Figure 5.14: LU Factorization speedups

All message passing implementations except MPJ/Ibis on TCP show the same speedup, though they do not scale perfectly. Due to a relatively small problem size the effect of the computation part becomes small. Because of memory constraints it was not possible to enlarge the problem size beyond 6000 x 6000 items.

Series This benchmark computes the first 10^6 Fourier coefficients of a function inside a predefined interval. Communication only takes place in the end of the computation of each node sending its individual results (double arrays) to node 0. The performance of both computation and communication is measured.

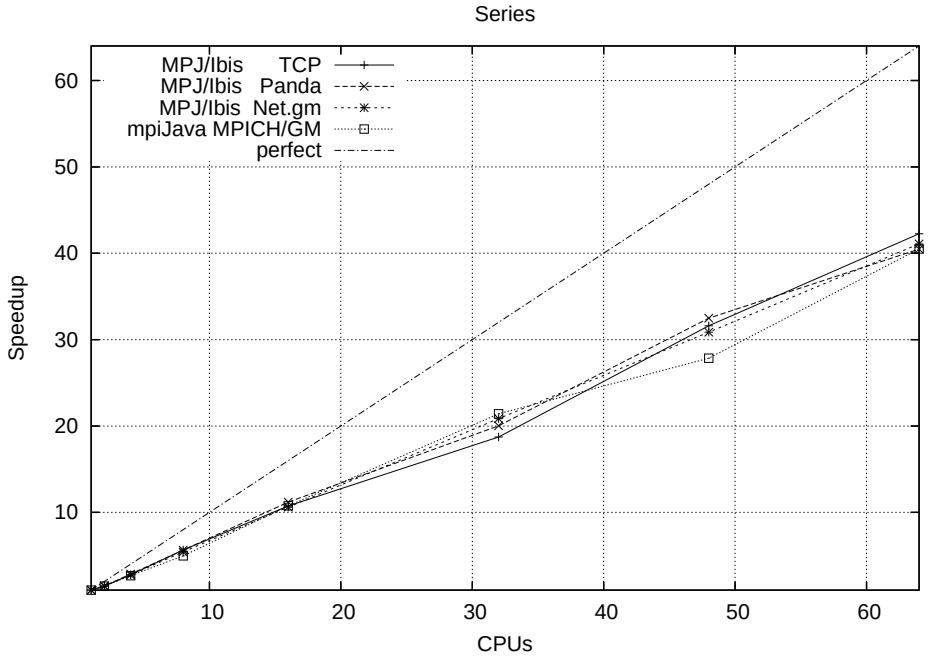

Figure 5.15: Series speedups

The speedups for all implementations grow in a linear way shown in Figure 5.15. But they are not perfect. At 48 nodes mpiJava is slightly slower than the Myrinet implementations of MPJ/Ibis. Even MPJ/Ibis on Fast Ethernet does not scale worse than the Myrinet implementations. This kernel executed at 64 nodes is 40 times faster than executed at only one node.

Sparse Matrix Multiplication This kernel multiplies a sparse matrix using one array of double values and two integer arrays. First, node 0 creates the matrix data and transfers it to each process. Second, when each node has completed the computation the result will be combined using allreduce (sum operation). Only the time of step two is measured. Here, a sparse matrix with a size of 10^6x 10^6 items has been used for 200 iterations. It was not possible to enlarge the matrix size, because of memory restrictions.

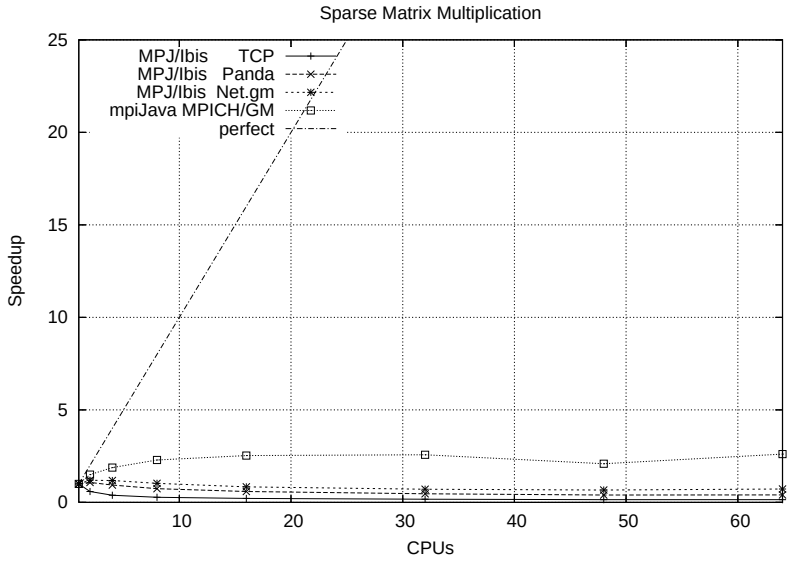

Figure 5.16: Sparse matrix multiplication speedups

This benchmark has shown the smallest run times of the whole JGF benchmark suite (about 131 seconds for execution at only one node). This means, that the computation time is marginally small and does not affect the overall execution speed. Therefore the communication overhead becomes the main factor, which is getting higher on a growing number of involved processes leading to performance reduction (see Figure 5.16). For all MPJ/Ibis implementations the speed is slowed down showing the impact of the higher latencies shown in Section 5.2. For mpiJava there is a small speedup of about 2,5. But it is to be noticed that one of MPICH/GMs *allreduce* operations is erroneous. Beyond eight participating processes this benchmark reports result validation errors for mpiJava. MPICH/GM uses different algorithms for *allreduce* depending on data size and the number of processes. Here, at more than

eight processes it switches to a different algorithm, which does not work correctly. The speedup values for this benchmark on mpiJava are not representative.

Successive Over-Relaxation This benchmark performs 100 iterations of successive over-relaxation (SOR) on a 6000 x 6000 grid. The arrays are distributed over processes in blocks using the red-black checkerboard ordering mechanism. Only neighbouring processes exchange arrays, which consist of double values. The arrays are treated as objects, since they are two-dimensional. It was not possible to run this benchmark with MPJ/Ibis on Net.gm, due to the mentioned problems in buffer reservation of Net.gm.

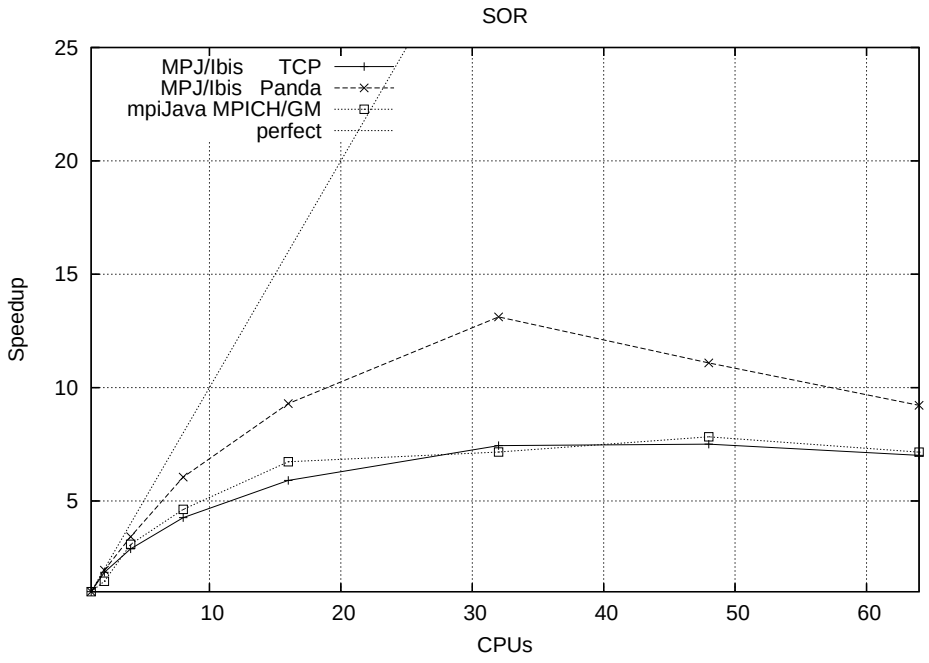

Figure 5.17: Successive over-relaxation speedups

Because of the Ibis serialization model MPJ/Ibis on Panda outperforms mpiJava by a factor of two until 32 participating processes (see Figure 5.17). Beyond 32 nodes the transmitted arrays become smaller reducing the performance advantage by increasing the relative communication overhead.

5.3.3 Section 3: Applications

Molecular Dynamics The Molecular Dynamics application models 27436 particles (problem size: 19) interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. In each iteration the particles are being updated using the *allreduce* operation with summation in the following way:

- three times with double arrays
- two times with a double value
- once with an integer value

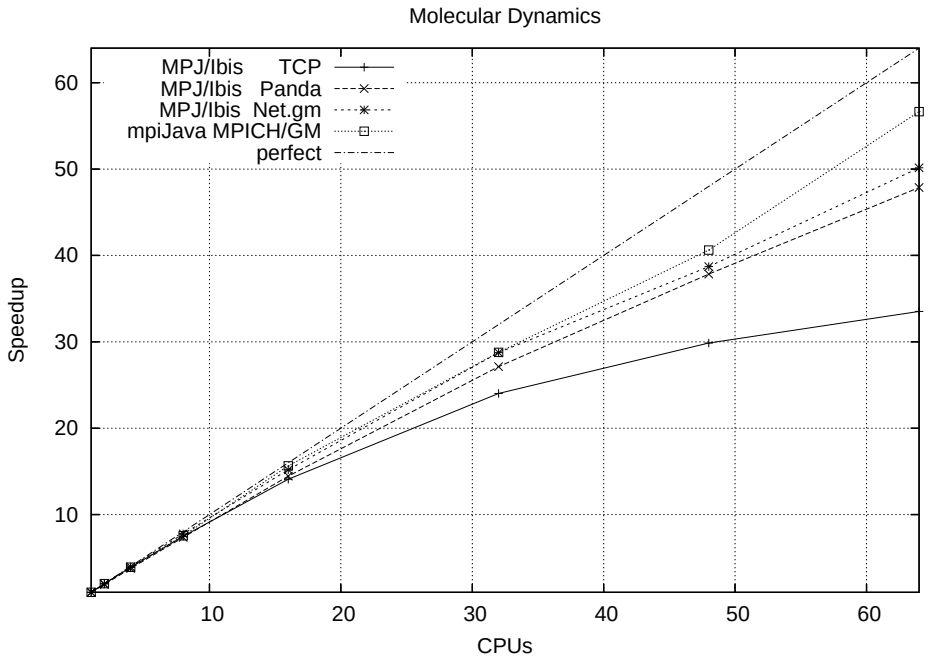

Figure 5.18: Molecular dynamics speedups

As can be seen in Figure 5.18 mpiJava slightly outperforms MPJ/Ibis on the Myrinet modules. In contrast to the sparse matrix multiplication this benchmark does not report validation errors with mpiJava, since MPICH/GM does not switch between different *allreduce* algorithms in this case.

Monte Carlo Simulation This financial simulation uses Monte Carlo techniques to price products derived from an underlying asset. It generates 60000 sample time series. The results at each node are stored in object arrays of class *java.util.Vector*, which are sent to node 0 using the *send* and *recv* primitives. As with the successive over-relaxation kernel it was not possible to run this benchmark using Net.gm for Ibis.

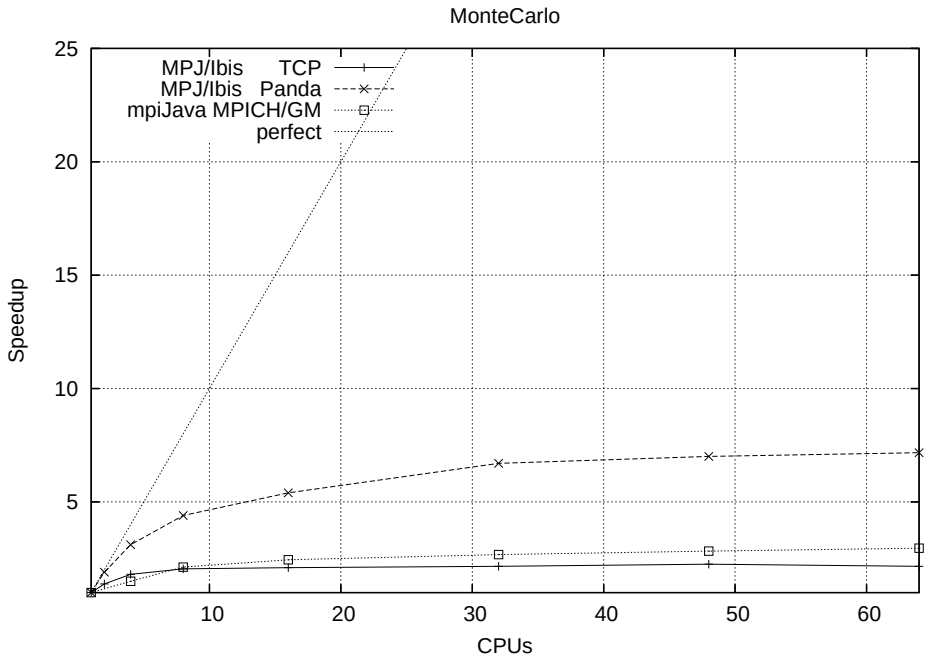

Figure 5.19: Monte Carlo speedups

Though none message passing implementation achieves perfect speedup, MPJ/Ibis on Panda outperforms mpiJava considerably (see Figure 5.19), since it takes advantage of the Ibis serialization mechanism. While mpiJava depends on Sun serialization, it is slightly faster than MPJ/Ibis on TCP, which is restricted by the available bandwidth of Fast Ethernet.

Raytracer The raytracer application benchmark renders a scene containing 64 spheres at a resolution of 2000 x 2000 pixels. Each process computes a part of the scene and sends the rendered pixels to node 0 using the *send* and *recv* primitives.

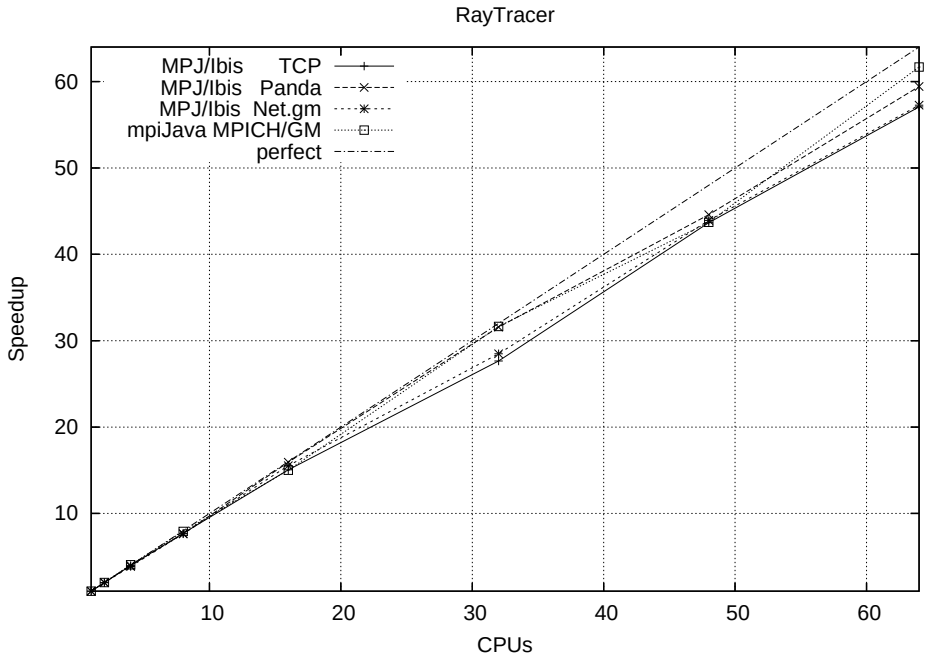

Figure 5.20: Raytracer speedups

In all cases the raytracer scales almost perfectly. At 64 nodes mpiJava shows a slightly better performance than MPJ/Ibis. In comparison to the computation part the communication overhead is marginally small for all message passing implementations evaluated.

5.4 Discussion

MPJ/Ibis and mpiJava have been evaluated both via micro benchmarks (Section 5.2) measuring latencies and throughputs and via the JGF benchmark suite (Section 5.3) measuring throughputs, the collective operations, kernel and application runtimes.

The micro benchmarks have shown that MPJ/Ibis does not come with a great performance lack caused by MPJ itself. In comparison to mpiJava on MPICH/GM MPJ/Ibis shows higher latencies, but the influence becomes smaller at growing data sizes. Particularly for object arrays, MPJ/Ibis has a great performance advantage over mpiJava on MPICH/GM.

On the other hand the effects of the relative higher latencies of MPJ/Ibis become visible at the low level benchmarks of the JGF benchmark suite in section 1. Particularly, the algorithm for the *barrier* operation should be improved. MpiJava performs better when basic type arrays are being communicated through the different collective operations, while MPJ/Ibis has advantages when object arrays are being transfered.

The performance lack of MPJ/Ibis for basic type arrays almost disappears at the sections 2 and 3, where more computation intensive applications are benchmarked. At kernels and applications based on object arrays MPJ/Ibis' speedups are considerably higher (see SOR and Monte Carlo). The only benchmarks of section 2 and 3, where mpiJava shows substantially higher speedups than MPJ/Ibis are the Sparse Matrix Multiplication and the Molecular Dynamics. In both applications *allreduce* is the main operation used for communication, where, as mentioned above, the correctness of MPICH/GM can be doubted.

The evaluation has shown that MPJ/Ibis' performance is highly dependent on the underlying Ibis implementation. In conclusion, MPJ/Ibis' performance is competitive to that of mpiJava. Additionally, MPJ/Ibis comes with an advantage of full portability and flexibility allowing MPJ/Ibis to run in heterogeneous networks in contrast to mpiJava, which depends on a native MPI implementation.

Chapter 6

Related Work

As mentioned in the beginning of Chapter 4 a lot of research has been done to develop an MPI binding to Java resulting in a variety of different implementations. MPJ/Ibis takes place in this history, which participants will be introduced in the following. All projects are being presented with a view to efficiency and portability.

JavaMPI JavaMPI [17] is based on various functions using JNI to wrap MPI methods to Java. For that purpose a Java-to-C Interface generator (JCI) has been implemented to create a C-stub function and a Java method declaration for each native method to be exported from the MPI library. The automatic wrapper creation resulted in an almost complete Java binding to MPI-1.1 [15] with less implementation costs. However, JavaMPI applications are not portable, since a native MPI implementation is always required for execution. The JavaMPI project is maintained by the University of Westminster, but no longer active. The last version[1] has been released in 2000.

jmpi Jmpi [6] implemented at Baskent University in 1998 works on top of JPVM [7]. Both jmpi and JPVM are implemented entirely in Java. JPVM follows the concept of parallel virtual machines (PVM). The main difference [9] between MPI and PVM is, that PVM is optimized for fault tolerance in heterogeneous networks using a small set of standard communication techniques (e.g. TCP). That allows

[1] JavaMPI online at http://perun.hscs.wmin.ac.uk/JavaMPI/

jmpi applications to be highly portable, but also limits communication performance dramatically. Here jmpi suffers from the poor performance of JPVM. The concepts of PVM will not discussed further at this point. The jmpi project is no longer maintained.

MPIJ The MPIJ [12] implementation is written in pure Java and runs as a part of the Distributed Object Group Metacomputing Architecture (DOGMA) [11] using RMI for communication. If available on the running platform, MPIJ additionally uses native marshaling of primitive types instead of Java marshaling. DOGMA[2] has been developed at Brigham Young University in 1998. Only a precompiled package of the current DOGMA implementation has been released, but due to almost non-existent documentation it could not determined, if the current DOGMA implementation still contains MPIJ.

JMPP JMPP [5] has been developed at the National Chiao-Tung University. In general this implementation is also built on top of RMI resulting in performance disadvantages. To achieve more flexibility an additional layer between the classes implementing the MPI methods and RMI has been implemented, called Abstract Device Interface (ADI). It abstracts completely from the underlying communication layer allowing to replace RMI with other modules for more efficient communication. Currently ADI only supports RMI. While the JMPP project is inactive, a more efficient implementation of ADI can not be expected.

JMPI Using RMI for communication JMPI [19] has been implemented entirely in Java with the advantage of full portability. Since RMI causes high performance loss, an optitmized RMI model called KaRMI [21] has been used for data transfer. KaRMI improves the performance of JMPI notably, but comes with reduction to portability, since it has to be configured explicitly for each different JVM used increasing administration overhead for each JMPI application. JMPI has been developed at the University of Massachusetts[3], but a release is not available.

[2] DOGMA online at http://csl.cs.byu.edu/dogma/
[3] http://www.umass.edu/

CCJ In contrast to the other projects CCJ[4] [20] implemented at Vrije Universiteit Amsterdam in 2003 follows a strict object-oriented approach and thus can not be claimed to be a binding to the MPI specification. CCJ also has been built directly on top of RMI with all the disadvantages that come along with it. Nevertheless group communication is possible, where threads within the same thread group can exchange messages using collective operations like *broadcast*, *gather* and *scatter*. *Alltoall* is not supported here. To reach higher communication speeds it is also possible for CCJ applications to be compiled with Manta[5] [14, 37-68], a native Java compiler optimized for RMI allowing remote method invocation on Myrinet based networks. Manta is source code compatible to Java version 1.1. While CCJ compiled with Manta works more efficiently, the use of a native compiler breaks Javas portability advantage. Both projects are no longer active.

mpiJava MpiJava[6] [1] is based on wrapping native methods like the MPI implementation MPICH with the Java Native Interface (JNI). The API is modeled very closely on the MPI-1.1 standard provided by the MPI Forum, but does not match the proposed MPJ [4] specification. MpiJava comes with a large set of documentation including a complete API reference. Since it is widely used to enable message passing for Java, it has been chosen to be compared with MPJ/Ibis (see Chapter 5). However, mpiJava comes with some notably disadvantages:

- compatibility issues with some native MPI implementations (e.g. MPICH/P4)
- reduced portability, since an existing native MPI library is needed for the target platform

This project still is active.

MPJ In 2004 the Distributed Systems Group at the University of Portsmouth announced a message passing implementation matching the MPJ specification. This project is also called MPJ[7], but a release is not publicly available. MPJ implements

[4] CCJ online at http://www.cs.vu.nl/ibis/ccj_download.html
[5] Manta online at http://www.cs.vu.nl/~robn/manta/
[6] mpiJava online at http://www.hpjava.org/mpiJava.html
[7] MPJ Project online at http://dsg.port.ac.uk/projects/mpj/

an MPJ Device layer, which abstracts from the underlying communication model. For TCP based networks it uses the *Java.nio* package and a wrapper class using JNI for Myrinet. Like MPJ/Ibis this implementation is completely written in Java, but currently supports only the point-to-point primitives, a small subset of the MPJ specification. However, some low level benchmark results are being presented on the projects website concerning basic type array transmission, with competitive results. The issue of efficient object serialization seems not to be cleared yet. Since the MPJ project is in early stage, more results have to be expected in the future.

Summary All of the Java message passing projects introduced in this chapter have shown disadvantages. Either an implementation is efficient, but does not benefit from Javas portability, or it is highly portable, but suffers from the poor performance of the underlying communication model. The only existing project that seems to provide efficiency and flexibility is MPJ, which still is under development for the first release and not publicly available at the moment.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

In this thesis a new message passing platform for Java called MPJ/Ibis has been presented. The main focus was to implement an environment that performance can compete with existing Java bindings of MPI (e.g. mpiJava), but without flexibility drawbacks.

Chapter 2 introduced parallel architectures in general and the basic principles of message passing derived from the MPI-1.1 specification. In summary the specification defines the following concepts:

- Point-to-point communication
- Groups of processes
- Collective communication
- Communication contexts
- Virtual topologies
- Derived datatypes

In Chapter 3 Javas drawbacks for parallel computation have been pointed out. Besides the great advantage of portability Java also shows disadvantages, particularly RMI is not flexible and efficient enough to meet the requirements of an efficient

message passing environment. The grid programming environment Ibis addresses these drawbacks (see Sections 3.2 and 3.3) and thus has been chosen for a message passing implementation to be built on top of. The main advantages of Ibis in short are:

- Efficient serialization
- Efficient communication

Additionally, Ibis' flexibility allows any MPJ/Ibis application to run on clusters and grids without recompilation by loading the appropriate communication module at runtime.

Putting Ibis and MPI together in Chapter 4 the proposed MPJ specification has been taken as basis, since it is the result of research within the Java Grande Forum and specifies a well defined API for a Java binding of MPI. Chapter 4 also focuses the main implementation details, particularly the point-to-point primitives, the collective operation algorithms and context management.

In Chapter 5 MPJ/Ibis has been evaluated using micro benchmarks and the benchmark suite for MPJ implementations provided by the Java Grande Forum. During evaluation for Myrinet networks MPJ/Ibis has been opposed mpiJava (see Chapter 6). The low level results have shown that MPJ/Ibis has great advantages over mpiJava, when objects have to be serialized, while mpiJava moderately outperforms MPJ/Ibis, when basic type arrays have to be communicated. For most kernels and applications used in the benchmarks, where relative communication overhead has been reduced, MPJ/Ibis and mpiJava have shown almost equivalent results.

The benchmarks have shown that flexibility provided by MPJ/Ibis does not come with considerable performance penalties. In summary, MPJ/Ibis can be considered as a message passing platform for Java that combines competitive performance with portability ranging from high-performance clusters to grids. It is the first known Java binding of MPI that provides both flexibility and efficiency.

7.2 Outlook

Showing the relevance of MPJ/Ibis for the message passing community parts of this thesis have found their way into a publication[1], which has been accepted at EURO PVM/MPI 2005 conference, Sorrento (Naples), Italy and will be printed in Lecture Notes of Computer Science (LNCS)[2]. Nevertheless, some tasks still exist for MPJ/Ibis to be done. Implementing the virtual topologies and improving the collective operations (ie. *barrier*) can be done in the near future.

The issue of supporting the whole set of methods needed for derived datatypes is not cleared yet. As pointed out in Section 4.5 it is not necessary to support derived datatypes in MPJ/Ibis, since Java objects support them natively. On the other hand the existence of derived datatypes would ease the work of software developers to port existing MPI applications written in C, Fortran or other languages to MPJ/Ibis. Thus, it should be figured out how the issue of multidimensional arrays in Java, which in fact is the main handicap for derived datatypes, can be addressed. In 2001 the Ninja [18] project (Numerically Intensive Java) supported by IBM proposed an extension to add truly multidimensional arrays to Java. Although the proposal has been withdrawn from Suns Java specification request program[3] and thus will not be added into future Java specifications, the ambitions of the Ninja approach should be continued.

Since MPJ/Ibis depends on the underlying Ibis implementations, Ibis and particularly Net.gm and Panda should be improved in future work to provide more stability to MPJ/Ibis. Furthermore, it should be investigated how the MPJ specification (and thus MPJ/Ibis) can be extended towards MPI-2. Paralleling single-sided communication and dynamic process management are additional interesting aspects especially for global grids, when fault tolerance becomes an issue.

[1] *MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java* online at: http://www.cs.vu.nl/ibis/papers/europvm2005.pdf

[2] B. Di Martino et al. (Eds.): *EuroPVM/MPI 2005*, LNCS Volume Number 3666, pp. 217-224, 2005, Springer Verlag Berlin Heidelberg 2005

[3] Java Specification Request:
Multiarray package online at http://jcp.org/en/jsr/detail?id=083

Bibliography

[1] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpijava: An object-oriented java interface to mpi. In *Presented at International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, San Juan, Puerto Rico, Apr. 1999. LNCS, Springer Verlag, Heidelberg, Germany.

[2] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. pages 213–226, 1993.

[3] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Java Grande*, pages 97–105, 2001.

[4] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[5] Y.-P. Chen and W. Yang. Java message passing package - a design and implementation of mpi in java. In *Proceedings of the Sixth Workshop on Compiler Techniques for High-Performance Computing*, Kaohsing, Taiwan, Mar. 2000.

[6] K. Dincer. Ubiquitous Message Passing Interface Implementation in Java: jmpi. In *IPPS/SPDP*, pages 203–211. IEEE Computer Society, 1999.

[7] A. Ferrari. JPVM: network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11–13):985–992, 1998.

[8] B. Goetz. *Threading Lightly: Synchronization is not the enemy*, online at ftp://www6.software.ibm.com/software/developer/library/j-threads1.pdf edition, 2001.

[9] W. Gropp and E. L. Lusk. Why are PVM and MPI so different? In *PVM/MPI*, pages 3–10, 1997.

[10] W. Huber. *Paralleles Rechnen*. Oldenbourg, München, 1997.

[11] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice and Experience*, 10:977–983, 1998.

[12] G. Judd, M. J. Clement, Q. Snell, and V. Getov. Design issues for efficient implementation of MPI in java. In *Java Grande*, pages 58–65, 1999.

[13] G. Krüger. *Go To Java 2*. Addison Wesley, 1999.

[14] J. Massen. *Method Invocation Based Communication Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, June 2003.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, online at http://www.mpi-forum.org/docs/mpi-11.ps edition, 1995.

[16] Message Passing Interface Forum. *MPI2: Extensions to the Message-Passing Interface*, online at http://www.mpi-forum.org/docs/mpi-20.ps edition, 1997.

[17] S. Mintchev and V. Getov. Towards portable message passing in java: Binding MPI. In *PVM/MPI*, pages 135–142, 1997.

[18] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, P. Wu, and G. Almasi. The NINJA project. *Communications of the ACM*, 44(10):102–109, 2001.

[19] S. Morin, I. Koren, and C. M. Krishna. JMPI: Implementing the Message Passing Standard in Java. In *IPDPS*, 2002.

[20] A. Nelisse, J. Maassen, T. Kielmann, and H. E. Bal. CCJ: object-based message passing and collective communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3-5):341–369, 2003.

[21] M. Philippsen and B. Haumacher. More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732, 1999.

[22] Sun Microsystems. *Java Remote Method Invocation Specification*, online at http://java.sun.com/products/jdk/rmi edition, July 2005.

[23] Sun Microsystems. *Object Serialization Specification*, online at http://java.sun.com/j2se/1.4.2/docs/guide/serialization/index.html edition, July 2005.

[24] A. S. Tanenbaum and J. Goodman. *Computer Architektur*. Pearson Studium, München, 2001.

[25] R. V. van Nieuwpoort. *Efficient Java-Centric Grid-Computing*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, Sept. 2003.

[26] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.