MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java

Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann

Vrije Universiteit, Amsterdam, The Netherlands
`http://www.cs.vu.nl/ibis`

Abstract. The MPJ programming interface has been defined by the Java Grande forum to provide MPI-like message passing for Java applications. In this paper, we present MPJ/Ibis, the first implementation of MPJ based on our Ibis programming environment for cluster and grid computing. By exploiting both flexibility and efficiency of Ibis, our MPJ implementation delivers high-performance communication, while being deployable on various platforms, from Myrinet-based clusters to grids. We evaluated MPJ/Ibis on our DAS-2 cluster. Our results show that MPJ/Ibis' performance is competitive to mpiJava on Myrinet and Fast Ethernet, and to C-based MPICH on Fast Ethernet.

1 Introduction

In recent years, Java has gained increasing interest as a platform for high performance and Grid computing [1]. Java's "write once, run anywhere" property has made it attractive, especially for high-performance grid computing where many heterogeneous platforms are used and where application portability becomes an issue with compiled languages like C++ or Fortran.

In previous work on our Ibis programming environment [2], we showed that parallel Java programs can run and communicate efficiently. Ibis supports object-based communication: method invocation on remote objects and object groups, as well as divide-and-conquer parallelism via spawned method invocations [2]. The important class of message-passing applications was not supported so far.

To enable message passing applications, the Java Grande Forum proposed MPJ [3], the MPI language bindings to Java. So far, no implementation of MPJ has been made available. In this paper, we present MPJ/Ibis, our implementation of MPJ on top of the Ibis platform. Being based in Ibis, MPJ/Ibis can be deployed flexibly and efficiently, 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.

In this paper, we discuss our design choices for implementing the MPJ API. As evaluation, we run both micro benchmarks and applications from the JavaGrande benchmark suite [1]. Micro benchmarks show that on a Myrinet cluster, MPJ/Ibis communicates slower than C-based MPICH, but outperforms MPI-Java, an older Java wrapper for MPI. Using TCP on Fast Ethernet shows that MPJ/Ibis is significantly faster than C-based MPICH. (Unfortunately, MPI-Java does not run at all in this configuration.) With the JavaGrande bench-

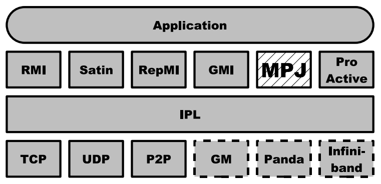


Fig. 1. Design of Ibis. The various modules can be loaded dynamically.

mark applications, MPJ/Ibis is either on-par with MPIJava or even outperforms it. MPJ/Ibis can thus be considered as a message-passing platform for Java that combines competitive performance with portability ranging from high-performance clusters to grids.

2 Related Work

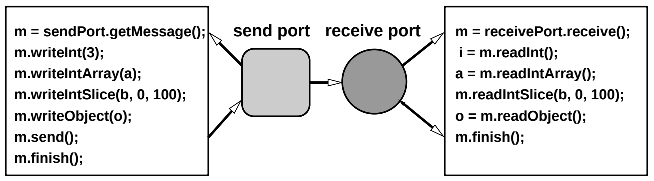
Many attempts were made to bind MPI to Java. MpiJava [4] is based on wrapping native methods like the MPI implementation MPICH with the Java Native Interface (JNI). The API is modeled very closely on the MPI standard provided by the MPI Forum. Due to limitations of the Java language (primitive type arguments cannot be passed as reference), small changes to the original standard had been made. JavaMPI [5] also uses JNI to wrap native methods to Java. It overcomes the argument passing problems using automatically generated C-stub functions and JNI method declarations. The MPIJ [6] implementation is written in pure Java and runs as a part of the Distributed Object Group Metacomputing Architecture (DOGMA) [7]. If available on the running platform, MPIJ uses native marshaling of primitive types instead of Java marshaling.

The first two approaches provide fast message passing, but do not match Java's "write once, run anywhere" property. JavaMPI and mpiJava are not portable enough, since a it requires a native MPI library and the Java binding must be compiled on the target system. MPIJ is written in Java and addresses the conversion of primitive datatypes into byte arrays. However, it does not solve the more general problem of Object serialization, which is a bottleneck.

MPJ [3] proposes MPI language bindings to Java. These bindings merge the earlier proposals mentioned above. In this paper, we present MPJ/Ibis, which is the first available implementation of MPJ. MPJ/Ibis features a pure Java implementation, but can also use high speed networks using some native code. Moreover, MPJ/Ibis uses Ibis' highly efficient object serialization, greatly speeding up the sending of complex data structures.

3 Ibis, Flexible and Efficient Grid Programming

Our MPJ implementation runs on top of Ibis [2]. The structure of Ibis is shown in Figure 1. A central part of the system is the Ibis Portability Layer (IPL) which



```
m = sendPort.getMessage();
m.writeInt(3);
m.writeIntArray(a);
m.writeIntSlice(b, 0, 100);
m.writeObject(o);
m.send();
m.finish();
```

```
m = receivePort.receive();
 i = m.readInt();
a = m.readIntArray();
m.readIntSlice(b, 0, 100);
o = m.readObject();
m.finish();
```

Fig. 2. Send ports and receive ports.

consists of a small number of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. The IPL defines both serialization (the conversion of objects to bytes) and communication. Ibis also provides more high-level programming models, see [2]. In this paper, we focus on the MPJ programming model.

A key problem in making Java suitable for grid programming is designing a system that obtains high communication performance while retaining Java's portability. Current Java runtime environments are heavily biased to either portability or performance. The Ibis strategy to achieve both goals simultaneously is to develop reasonably efficient solutions that work "anywhere", supplemented with highly optimized solutions for increased performance in special cases. With Ibis, grid applications can run simultaneously on a variety of different machines, using optimized software where possible (e.g., Myrinet), and using standard software (e.g., TCP) when necessary.

3.1 Send Ports and Receive Ports

The IPL provides communication primitives using send ports and receive ports. A careful design of these ports and primitives allows flexible communication channels, streaming of data, efficient hardware multicast and zero-copy transfers. The layer above the IPL creates send and receive ports, which are connected to form a *unidirectional message channel*, see Figure 2. New (empty) message objects can be requested from send ports, and data items of any type can be inserted. Both primitive types and arbitrary objects can be written. When all data is inserted, the *send* primitive can be invoked on the message.

The IPL offers two ways to receive messages. First, messages can be received with the receive port's blocking *receive* primitive (see Figure 2). It returns a message object, from which the data can be extracted using the provided set of read methods. Second, the receive ports can be configured to generate *upcalls*, thus providing the mechanism for implicit message receipt. An important insight is that zero-copy can be made possible in some important special cases by carefully designing the port interfaces. Ibis allows native implementations to support zero-copy for array types, while only one copy is required for object types.

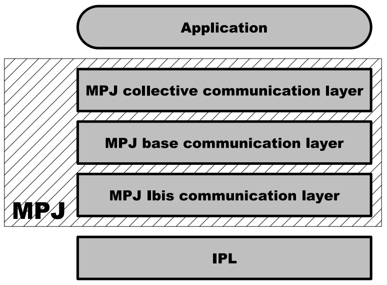

Application
MPJ collective communication layer
MPJ base communication layer
MPJ Ibis communication layer
MPJ
IPL

Fig. 3. Design of MPJ/Ibis.

3.2 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, Ibis can communicate through firewalls, even without explicitly opened ports.

The Myrinet implementation of the IPL is built on top of the native GM 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.

4 MPJ/Ibis

MPJ/Ibis is written completely in Java on top of the Ibis Portability Layer. It matches the MPJ specification mentioned in [3]. The architecture of MPJ/Ibis, shown in Figure 3, is divided into three layers. The *Communication Layer* provides the low level communication operations. The *MPJObject* class stores MPJ/Ibis messages and the information needed to identify them, ie. tag and context id. To avoid serialization overhead the *MPJObject* is not sent directly, but is split into a header and a data part. When header and message arrive at the destination, MPJ/Ibis decides either to put the message directly into the receive buffer or into a queue, where the retrieved message waits for further processing.

The *Base Communication Layer* takes care of the basic sending and receiving operations in the MPJ specification. It includes the blocking and nonblocking *send* and *receive* operations and the various *test* and *wait* statements. It is also responsible for group and communicator management. The *Collective Communication Layer* implements the collective operations on top of the *Base Communication Layer*. The algorithms realizing the collectives are shown in table 1.

Table 1. Algorithms used in MPJ to implement the collective operations.

Collective Operation	Algorithm
allgather	double ring
allgatherv	ring
alltoall	flat tree
alltoallv	flat tree
barrier	flat tree
broadcast	binomial tree
gather	flat tree
gatherv	flat tree
reduce	binomial tree
reduceScatter	phase 1:reduce; phase 2: scatterv
scan	flat tree
scatter	flat tree
scatterv	flat tree

4.1 MPJ/Ibis Implementation

MPJ/Ibis tries to avoid expensive operations like buffer copying, serialization and threads where it is possible. On the sender side, MPJ/Ibis analyses the message to find out if there is a need to copy it into a temporary buffer. This is necessary when using displacements, for example. If no copy is required, the message will be written directly to the Ibis send port.

On the receiver side MPJ/Ibis has to decide to which communicator the message is targeted. The receive operation uses a blocking downcall receive to the Ibis receive port, where it waits for a message to arrive. When the message header comes in MPJ/Ibis determines if this message was expected. If it was not (a rare event), the whole message including the header will be packed into a MPJObject and then moved into a queue, copying then is mandatory. Otherwise MPJ/Ibis decides either to receive the message directly into the user's receive buffer or into a temporary buffer from where it will be copied to it's final destination (when displacements are used, for instance). There is no need to use threads for the blocking send and receive operations in MPJ/Ibis, which saves a lot of processor time. In many simple but often occurring cases zero-copying is possible as well. MPJ supports non-blocking communication operations, such as *isend* and *irecv*. These are built on top of the blocking operations using Java threads.

4.2 Open Issues

Since Java provides derived datatypes natively 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, which need at least a pair of values inside a given one-dimensional array. At the moment MPJ supports one-dimensional arrays. Multidimensional arrays can be sent as an object. In place receive is not possible in this case. MPJ/Ibis supports creating and splitting of new communicators, but intercommunication is not implemented yet. At this moment, MPJ/Ibis does not support virtual topologies.

Table 2. Low-level performance. Latencies in microseconds, throughputs in MByte/s.

network / implementation	Myrinet			Fast Ethernet		
	latency	array throughput	object throughput	latency	array throughput	object throughput
MPICH / C	22	178	N.A.	1269	10.6	N.A.
mpiJava / SUN JVM	84	86	1.2	N.A.	N.A.	N.A.
mpiJava / IBM JVM	41	178	2.7	N.A.	N.A.	N.A.
Ibis IPL / SUN JVM	56	80	4.8	146	11.2	3.0
Ibis IPL / IBM JVM	46	128	12.8	144	11.2	4.4
MPJ / SUN JVM	98	80	4.6	172	11.2	3.0
MPJ / IBM JVM	58	128	12.4	162	11.2	4.4

5 Evaluation

We evaluated MPJ/Ibis on the DAS-2 cluster in Amsterdam, which consists of 72 Dual Pentium-III nodes with 1 GByte RAM, connected by Myrinet and Fast Ethernet. The operating system is Red Hat Enterprise Linux with kernel 2.4.

5.1 Low-Level Benchmarks

Table 2 shows low-level benchmark numbers for the IPL, MPJ/Ibis, MPICH and mpiJava. For the Java measurements, we used two different JVMs, one from Sun and one from IBM, both in version 1.4.2. For C, we used MPICH/GM for Myrinet and MPICH/P4 for Fast Ethernet. MpiJava uses MPICH/GM, we were unable to run it with MPICH/P4. First, we measured the roundtrip latency by sending one byte back and forth. On Myrinet, Java has considerably higher latencies than C. This is partly caused by switching from Java to C using the JNI. On Fast Ethernet MPJ is faster than MPICH/P4 (the latency is more than 7 times lower). In this case, only Java code is used, the JNI is not involved.

Next, we measured the throughput for 64 KByte arrays of doubles. The data is received in preallocated arrays, no new objects are allocated and no garbage collection is done by the JVM. The numbers show that the IBM JVM is much faster than the SUN JVM in this case, because the SUN JVM makes a copy of the array when going through the JNI. This almost halves the throughput. When we compare the mpiJava results on the IBM JVM and Myrinet with MPICH, we see that performance is the same. Ibis and MPJ are somewhat slower, but still achieve 128 MByte/s. On Fast Ethernet, all Java implementations are able to fill the network. MPICH/P4 is marginally slower.

Finally, we use a throughput test that sends binary trees of 1023 nodes, with four integer values payload per node. We show the throughput of the payload. In reality, more data is sent, such as type information and the structure of the tree (pointers to the left and right children). The tree is reconstructed at the receiving side, in newly allocated objects. It is not possible to express this test in C in this way. Ibis and MPJ are much more efficient than mpiJava when sending objects, resulting in a 4.5 times higher throughput, thanks to Ibis' highly efficient serialization implementation. This result is significant, because in Java programs typically send complex graphs of objects.

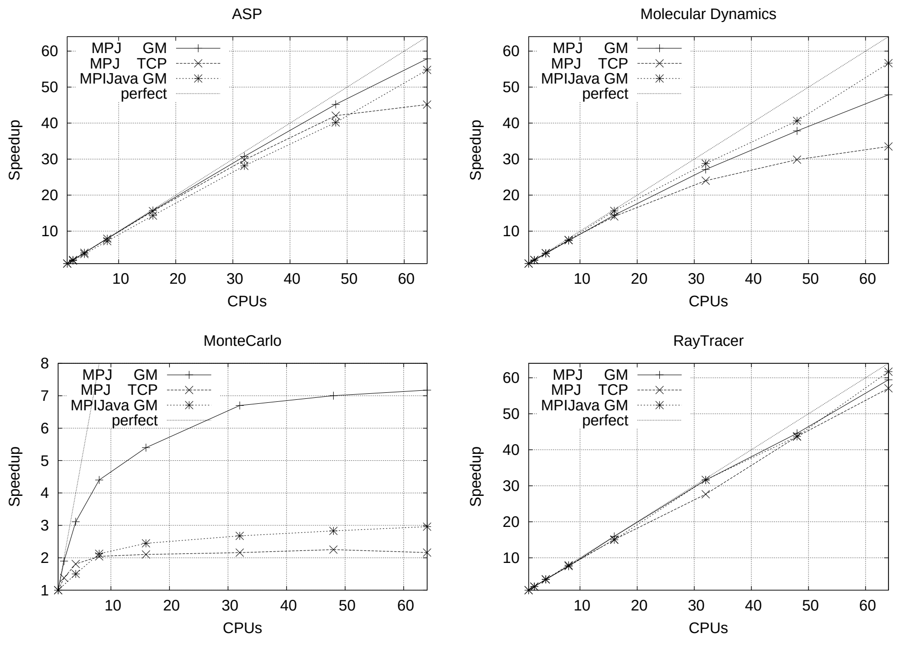


Fig. 4. Speedup of MPJ/Ibis and MPIJava applications.

5.2 Application Performance

Figure 4 shows the speedups achieved with three applications from the Java Grande MPJ benchmarks using the Sun JVM (we found that mpiJava is unstable in combination with the IBM JVM). We also show an additional application, ASP, which is not part of the Java Grande set.

ASP All-pairs Shortest Paths (ASP) computes the shortest path between any two nodes of a given 5952-node graph. In ASP one machine broadcasts an array of data each iteration. Both the MPJ/Ibis and mpiJava obtain excellent speedups, but MPJ/Ibis scales better to larger number of CPUs.

MolDyn is an N-body code. For each iteration, six reduce-to-all summation operations update the atoms. We enlarged the problem size to 19 (27436 particles). Both MPJ/Ibis and mpiJava perform well on this application. Only on 64 machines, mpiJava slightly outperforms MPJ/Ibis.

The **MonteCarlo** application is a financial simulation. Each node generates an array of Vector objects. These arrays of complex objects are sent to CPU 0 by individual messages. We cannot make the problem larger than size B due to memory constraints. With this problem size, neither mpiJava nor MPJ/Ibis scale well. However, MPJ/Ibis clearly is more efficient: it outperforms mpiJava with more than a factor of two, thanks to Ibis' highly efficient serialization mechanism.

Ray Tracer renders a scene of 64 spheres. Each node calculates a checksum over its part of the scene, and a reduce operation is used to combine these checksums into a single value. The machines send the rendered pixels to machine 0 by individual messages. We enlarged the problem to an image of 2000x2000 pixels. MPJ/Ibis and mpiJava perform almost perfectly on this application.

The measurements in this section show that MPJ/Ibis achieves similar performance as mpiJava. In one case (MonteCarlo), MPJ/Ibis outperforms mpiJava by a large margin. The results indicate that the flexibility provided by the MPJ implementation on top of Ibis does not come with a performance penalty.

6 Conclusions

We presented MPJ/Ibis, our implementation of the Java language binding of MPI. Our implementation is based on our Ibis grid programming environment. Putting a message-passing layer like MPJ on top of Ibis provides an efficient environment, allowing message-passing applications in Java. Ibis' flexibility then allows to run these applications on clusters and on grids, without recompilation, merely by loading the respective communication substrate at run time.

We have evaluated MPJ/Ibis using micro benchmarks and applications from the JavaGrande benchmark suite. Our results show that MPJ/Ibis shows competitive or better performance than MPIJava, an older MPI language binding. Comparing to C-based MPICH, MPJ/Ibis is somewhat slower using Myrinet, but outperforms its competitor when using TCP/IP over Fast Ethernet.

To summarize, 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. We are currently investigating the use of both MPJ and shared-object communication, paralleling single-sided communication as introduced in MPI-2.

References

1. The JavaGrande Forum: www.javagrande.org (1999)
2. van Nieuwpoort, R.V., Maassen, J., Hofman, R., Kielmann, T., Bal, H.E.: Ibis: an Efficient Java-based Grid Programming Environment. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, USA (2002) 18–27
3. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience **12** (2000) 1019–1038
4. Baker, M., Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava: An Object-Oriented Java interface to MPI. In: Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP, LNCS, Springer Verlag, Heidelberg, Germany (1999)
5. Mintchev, S., Getov, V.: Towards portable message passing in Java: Binding MPI. In: Recent Advances in PVM and MPI. Number 1332 in Lecture Notes in Computer Science (LNCS), Springer-Verlag (1997) 135–142
6. Judd, G., Clement, M., Snell, Q., Getov, V.: Design issues for efficient implementation of mpi in java. In: ACM 1999 Java Grande Conference. (1999) 58–65
7. Judd, G., Clement, M., Snell, Q.: DOGMA: Distributed Object Group Metacomputing Architecture. Concurrency: Practice and Experience **10** (1998) 977–983