Ibis: an Efficient Java-based Grid Programming Environment

Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, Henri E. Bal
Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{rob,jason,rutger,kielmann,bal}@cs.vu.nl http://www.cs.vu.nl/manta

ABSTRACT

In computational grids, performance-hungry applications need to simultaneously tap the computational power of multiple, dynamically available sites. The crux of designing grid programming environments stems exactly from the dynamic availability of compute cycles: grid programming environments (a) need to be *portable* to run on as many sites as possible, (b) they need to be *flexible* to cope with different network protocols and dynamically changing groups of compute nodes, while (c) they need to provide *efficient* (local) communication that enables high-performance computing in the first place.

Existing programming environments are either portable (Java), or they are flexible (Jini, Java RMI), or they are highly efficient (MPI). No system combines all three properties that are necessary for grid computing. In this paper, we present Ibis, a new programming environment that combines Java's "run everywhere" portability both with flexible treatment of dynamically available networks and processor pools, and with highly efficient, object-based communication. Ibis can transfer Java objects very efficiently by combining streaming object serialization with a zero-copy protocol. Using RMI as a simple test case, we show that Ibis outperforms existing RMI implementations, achieving up to 9 times higher throughputs with trees of objects.

1. INTRODUCTION

Computational grids can integrate geographically distributed resources into a seamless environment [8]. In one important grid scenario, performance-hungry applications use the computational power of dynamically available sites. Here, compute sites may join and leave ongoing computations. The sites may have heterogeneous architectures, both for the processors and for the network connections. Running high-performance applications on such dynamically changing platforms causes many intricate problems. The biggest challenge is to provide a programming environment and a runtime system that combine highly efficient execution and communication with the flexibility to run on dynamically changing sets of heterogeneous processors and networks.

Although efficient message passing libraries (e.g., MPI) are widely used, they were not designed for grid environments. MPI only marginally supports malleable applications that can cope with dynamically changing sets of processors. MPI implementations also have difficulties to efficiently utilize multiple, heterogeneous networks simultaneously; let alone switching between them at run time. For grid computing, more flexible, but still efficient, communication models and implementations are needed.

Many researchers believe that Java will be a useful technology to reduce the complexity of grid application programming [10]. Based on a well-defined virtual machine and class libraries, Java is inherently more portable than languages like C and Fortran, which are statically compiled in a traditional fashion. Java allows programs to run on a heterogeneous set of resources without any need for recompilation or porting. Modern JVMs such as the IBM JIT[1] or Sun Hotspot[2] obtain execution speed that is competitive to languages like C or Fortran [4]. Other strong points of Java for grid applications include type safety and integrated support for parallel and distributed programming. Java provides Remote Method Invocation (RMI) for transparent communication between JVMs.

Unfortunately, a hard and persistent problem is Java's inferior communication speed. In particular, Java's RMI performance is much criticized [3]. The communication overhead can be one or two orders of magnitude higher than that of lower-level models like MPI or RPC [19]. In earlier research on RMI [19], we have solved this performance problem by using a native compiler (Manta), replacing the standard serialization protocol by a highly efficient, compiler-generated zero-copy protocol written in C. Unfortunately, this approach fails for grid programming, as it requires a custom Java runtime system that cannot be integrated with standard JVMs. Thus, an important research problem is how to obtain good communication performance for Java without resorting to techniques that give up the advantages of its virtual machine approach.

In this paper, we present a Java-based grid programming environment, called *Ibis*, that allows highly efficient communication in combination with any JVM. Because Ibis is Java-based, it has the advantages that come with Java, such as portability, support for heterogeneity and security. Ibis has been designed to combine highly efficient communication with support for both heterogeneous networks and malleability. Ibis can be configured dynamically at run time, allowing to combine standard techniques that work "everywhere" (e.g., over TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect.

As a test case for our strategy, we implemented an optimized RMI system on top of Ibis. We show that, even on a regular JVM without any use of native code, our RMI implementation outperforms previous RMI implementations. When special native im-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1] see www.java.ibm.com
[2] see www.java.sun.com

plementations of Ibis are used, we can run RMI applications on fast user-space networks (e.g., Myrinet), and achieve performance that was previously only possible with special native compilers and communication systems.

In the remainder of this paper, we first present the design of the Ibis system (Section 2). Section 3 explains the Ibis implementation. We present a case study of Ibis RMI in Section 4. Related work is discussed in Section 5. Finally, we draw our conclusions in Section 6.

2. IBIS DESIGN

For deployment on the grid, it is imperative that Ibis be an extremely flexible system. Ibis should be able to provide communication support for any grid application, from the broadcasting of video to massively parallel applications. It should provide a unified framework for reliable and unreliable communication, unicasting and multicasting of data, and should support the use of any underlying communication protocol (TCP/IP, UDP, GM, etc.) Moreover, Ibis should support malleability (i.e., machines must be able to join and leave a running computation), even though some underlying communication systems may have a closed world assumption. Below, we will describe how Ibis was designed to support the aforementioned flexibility, while still achieving high performance.

Ibis consists of a runtime system, and a bytecode rewriter. The runtime system implements the API of the *Ibis Portability Layer* (IPL). The bytecode rewriter is used to generate bytecode for application classes to actually use the IPL, a role similar to the `rmic` of Sun RMI.

2.1 Design Overview of the Ibis Architecture

A key problem in making Java suitable for grid programming is how to design a system that obtains high communication performance while still adhering to Java's "write once, run everywhere" model. Current Java implementations are heavily biased to either portability or performance, and fail in the other aspect. Our strategy to achieve both goals simultaneously is to develop reasonably efficient solutions using standard techniques that work "everywhere", supplemented with highly optimized but non-standard solutions for increased performance in special cases. We apply this strategy to both computation and communication. Ibis is designed to use any standard Java Virtual Machine (JVM), but if a native optimizing compiler (e.g., Manta [19]) is available for a target machine, Ibis can use it instead. Likewise, Ibis can use standard communication protocols (e.g., TCP/IP or UDP, as provided by the JVM), but it can also plug in an optimized low-level protocol for a high-speed interconnect (e.g., GM or MPI), if available. We essentially aim to reuse all good ideas from the Manta native Java system, but now in pure Java. This is non-trivial, because pure Java lacks pointers, information on object layout, low level access to the thread package used inside the JVM, interrupts, and a `select` mechanism to monitor the status of a set of sockets. The challenges for Ibis are:

1. how to make the system so flexible that it can incorporate such a variety of software in a seamless way;
2. how to make the standard, 100% pure Java case efficient enough to be useful for grid computing;
3. study which additional optimizations can be done to improve performance further in special (high-performance) cases.

A grid application using Ibis can thus run on a variety of different machines, some using standard software and some using optimized software (a native compiler, the GM Myrinet protocol, MPI, etc.). Below, we discuss the three aforementioned issues in more detail.

2.1.1 *Flexibility*

The key characteristic of Ibis is its extreme flexibility, which is required to support grid applications. A major design goal is the ability to seamlessly plug in different communication substrates without changing the user code. For this purpose, the Ibis design uses the *Ibis Portability Layer* (IPL), which consists of a number of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. The layer on top of the IPL can negotiate with the IPL implementation through the well-defined IPL interface, and select the modules it needs. This flexibility is implemented using Java's dynamic class-loading mechanism. Although this kind of flexibility is very hard to achieve with traditional programming languages, it is relatively straightforward in Java.

Many message passing libraries such as MPI and GM guarantee reliable message delivery and FIFO message ordering. When applications do not require these properties, a different message passing library might be used to avoid the overhead that comes with reliability and message ordering. Using user-definable properties (key-value tuples), the IPL supports both reliable and unreliable communication, ordered and unordered messages, using a single, simple interface. This way, applications can create exactly the communication channels they need, without unnecessary overhead.

2.1.2 *Optimizing the Common Case*

To obtain acceptable communication performance, Ibis implements several optimizations. Most importantly, the overhead of serialization and reflection is avoided by compile-time generation of special methods (in bytecode) for each object type. These methods can be used to convert objects to bytes (and vise-versa), and create new objects on the receiving side, without using expensive reflection mechanisms. Another major source of overhead is in the JNI (Java Native Interface) calls required to convert the basic data types into byte arrays (and back). We have discovered that this problem can be solved by serializing into multiple buffers, one for each primitive data type. This optimization dramatically reduces the overhead of serialization.

Furthermore, our communication implementations use an optimized wire protocol. The Sun RMI protocol, for example, resends type information for each RMI. Our implementation, however, caches this type information per connection. Using this optimization, our protocol sends less data over the wire, but more importantly, saves processing time for encoding and decoding the type information.

2.1.3 *Optimizing Special Cases*

In many cases, the target machine may have additional facilities that allow faster computation or communication that is difficult to achieve with standard Java techniques. One example we investigated in previous work [19] is using a native, optimizing compiler instead of a JVM. This compiler (Manta) or any other high performance Java implementation from elsewhere can simply be used by Ibis. We therefore focus on optimizing communication performance. The most important special case for communication is the presence of a high-speed local interconnect. Usually, specialized user-level network software is required for such interconnects, instead of standard protocols (TCP, UDP) that use the OS kernel. Ibis therefore was designed to allow other protocols to be plugged in. So, lower-level communication may be based, for example, on a locally-optimized MPI library. We have developed low-level network software based on the Panda library [2], which can likewise be used by Ibis. Panda is a portable communication substrate, which has been implemented on a large variety of platforms and networks,

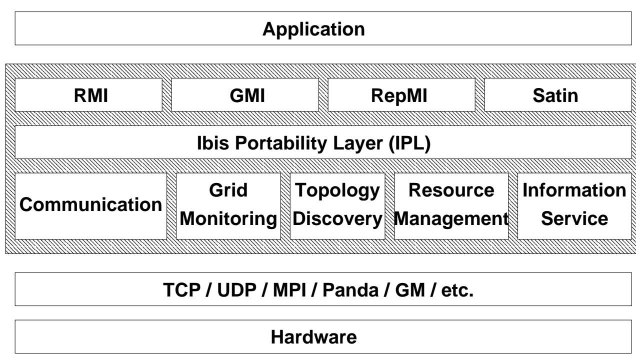

Figure 1: Design of Ibis. The various modules can be loaded dynamically, using run time class loading.

such as Fast Ethernet (on top of UDP) and Myrinet (on top of GM).

An important issue we study in this paper is the use of *zero-copy* protocols for Java. Such protocols try to avoid the overhead of memory copies, as these have a relatively high overhead with fast gigabit networks, resulting in decreased throughputs. With the standard serialization method used by most Java communication systems (e.g. RMI), a zero-copy implementation is impossible to achieve, since data is always serialized into intermediate buffers. With specialized protocols using Panda or MPI, however, a zero-copy protocol is at least possible for messages that transfer array datastructures. For graph datastructures, the number of copies can be reduced to one. Implementing zero-copy (or single-copy) communication in Java is a nontrivial task, but it is essential to make Java competitive with systems like MPI for which zero-copy implementations already exist. The zero-copy Ibis implementation is described in more detail in Section 3.2.

2.1.4 Design Overview

The overall structure of the Ibis system is shown in Figure 1. An important component is the IPL or *Ibis Portability Layer*, which consists of a set of Java interfaces that define how the layers above the IPL can make use of the lower Ibis components, such as communication and resource management. Because the components above the IPL can only access the lower modules via the IPL, it is possible to have multiple implementations of the lower modules. The IPL design will be explained in more detail in Section 2.2.

Generally, applications will not be built directly on top of the IPL (although this is possible). Instead, applications use some programming model for which an Ibis implementation exists. At this moment, we have implemented four runtime systems for programming models on top of the IPL: RMI, GMI, RepMI and Satin. RMI is Java's equivalent of RPC. However, RMI is object oriented and more flexible, as it supports polymorphism [28]. In [27] we showed that parallel grid applications can be written with RMI. Application-specific wide-area optimizations are needed, however. GMI [18] extends RMI with collective operations. GMI also uses an object-oriented programming model, and cleanly integrates into Java, as opposed to Java MPI implementations. We intend to port the wide-area optimizations we implemented in earlier work on MPI (MagPIe [14]), in order to make GMI more suitable for grid environments. RepMI extends Java with efficient replicated objects. In [17] we show that RepMI also works efficiently on wide-area systems. Satin [26] provides divide-and-conquer and replicated worker programming models, and is specifically targeted at grid systems. The four mentioned programming models are integrated into one single system, and can be used simultaneously. In this paper, we use our Ibis RMI implementation as a case study. Due to space limitations, we cannot discuss the other programming models. Below the IPL are the modules that implement the actual Ibis functionality, such as communication, and typical grid computing requirements such as performance monitoring and topology discovery.

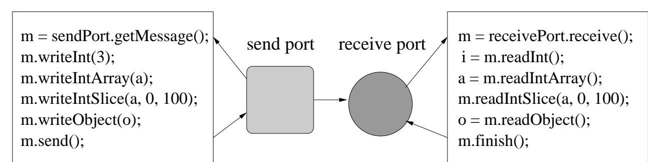

Figure 2: Send ports and receive ports.

2.2 Design of the Ibis Portability Layer (IPL)

The Ibis Portability layer is the interface between Ibis implementations for different architectures and the runtime systems that provide programming model support to the application layer. The IPL is a set of Java interfaces (i.e., an API), and does not contain any actual code. The philosophy behind the design of the IPL is the following: when efficient hardware primitives are available, make it possible to use them. Great care has to be taken to ensure that the use of mechanisms such as zero-copy protocols and hardware multicast are not made impossible by the interface. We will now describe the concepts behind the IPL and we will explain the design decisions.

2.2.1 Negotiating with Ibis Implementations

Ibis implementations are loaded at run time. Ibis allows the application or runtime system on top of it to negotiate with the available implementations, in order to select the implementation that best matches the specified requirements. More than one Ibis implementation can be loaded simultaneously, for example to provide gateway capabilities between different networks. For example, it is possible to load both a Panda Ibis implementation that runs on top of a Myrinet network for efficient communication inside a cluster, and a TCP/IP implementation for (wide-area) communication between clusters.

In contrast to many message passing systems, the IPL has no concept of hosts or threads, but uses location independent *Ibis identifiers* instead. The communication interface is object oriented. Applications using the IPL can create communication channels between objects, regardless of the location of the objects. The connected objects can be located in the same thread, on different threads on the same machine, or they could be located at different ends of the world. A registry is provided to locate communication endpoints using Ibis identifiers.

2.2.2 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, and zero-copy transfers. Therefore, the send and receive ports are important concepts in Ibis, and we will explain them in more detail below. The layer above the IPL can create send and receive ports, which are then connected (the send port initiates the connection) to form a *unidirectional message channel*. This process is shown in Figure 2. New (empty) message objects can be requested from send ports. Next, data items of any type can be inserted in this message. Both

primitive types such as *int*, *long* and *double*, and objects such as *String* or user-defined data types can be written. When all data has been inserted, the *send* primitive can be invoked on the message, sending it off.

The IPL offers two ways to receive messages. First, messages can be received with the receive port's blocking *receive* primitive (see Figure 2). The *receive* method returns a new message object, and the data can be extracted from the message 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. The upcall provides the message that has been received as a parameter. The data can be read with the same read methods described above. The upcall mechanism is provided in the IPL, because it is hard to implement an upcall mechanism efficiently on top of an explicit receive mechanism. Many applications and runtime systems rely on efficient implementations of upcalls (e.g., RMI, GMI and Satin). To avoid thread creation and switching, the IPL defines that there is at most *one* upcall thread running at a time per receive port.

Many message passing systems (e.g., MPI, Panda) are connectionless. Messages are sent to their destination, without explicitly creating a connection first. The IPL, however, provides a *connection-oriented* scheme (send and receive ports must be connected in order to communicate). This way, message data can be streamed. When large messages have to be transferred, the building of the message and the actual sending can be done simultaneously. This is especially important when sending large and complicated data structures, as can be done with Java serialization, because this incurs a large host overhead. It is thus imperative that communication and computation are overlapped.

A second design decision is to make the connections unidirectional. This is essential for the flexibility of the IPL, because it sometimes is desirable to have different properties for the individual channel directions. For example, when video data is streamed, the control channel from the client to the video server should be reliable. The return channel, however, from the video server back to the client should be an unreliable channel with low jitter characteristics. For some applications there is no return channel at all (e.g., wireless receivers that do not have a transmitter). The IPL can support all these communication requirements. Furthermore, on the internet, the outbound and return channel may be routed differently. It is therefore sensible to make it possible to export this to the layers above the IPL when required. Differences between the outbound and return channels are important for some adaptive applications or runtime systems, such as the Satin runtime system. Moreover, the IPL extends the send and receive port mechanism with multicast and many to one communication, for which unidirectional channels are more intuitive.

Another important insight is that zero copy can be made possible in some cases by carefully designing the interface for reading data from and writing data to messages. As can be seen in Figure 2, the IPL provides special read and write methods for all primitive arrays, and for slices of primitive arrays. The standard Java streaming classes (used for serialization and for writing to TCP/IP sockets) only provide a special case for byte arrays, and do not support array slices at all. All other datastructures, including the primitive types are converted to bytes first, even if the byte ordering of the sender and receiver is the same. When a pure Java implementation is used, the copying of the data is thus unavoidable. However, with this interface, we allow efficient native implementations to support zero copy for the array types, while only one copy is required for object types.

Connecting send ports and receive ports, creating a unidirectional channel for messages is the *only* communication primitive that the IPL provides. All other communication patterns can be constructed on top of this model. By creating both a send and receive port on the same host, bidirectional communication can be achieved, for example to implement an RPC-like system, as is shown in Figure 3. The IPL also allows a single send port to connect to multiple receive ports. Messages that are written to a send port that has multiple receivers are *multicast* (see Figure 4). Furthermore, multiple send ports can connect to a single receive port, thus implementing many-to-one communication (also shown in Figure 4).

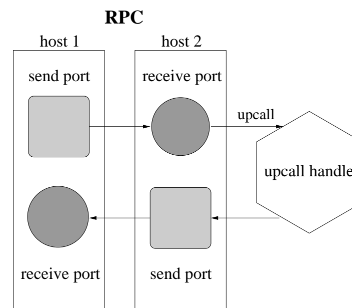

Figure 3: An RPC implemented with the IPL primitives.

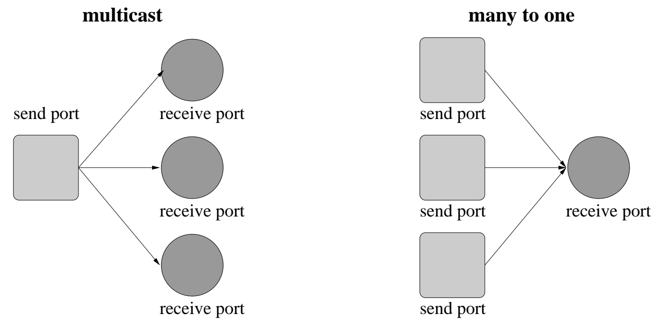

Figure 4: Other IPL communication patterns.

2.2.3 Port Types

All send and receive ports that are created by the layers on top of the IPL are *typed*. Port types are defined and configured via properties (key-value tuples) by the IPL user. Only ports of the same type

```
StaticProperties p = new StaticProperties();
p.add("message ordering", "FIFO");
p.add("reliability", "true");
try {
    PortType type = ibis.createPortType("port type", p);
} catch (IbisException e) {
    // Type creation fails when the Ibis implementation
    // does not provide the requested features.
}
try {
    SendPort s = type.createSendPort("my send port");
} catch (IbisException e) {
    // Handle errors.
}
```

Figure 5: Creating and configuring a new port type.

```
class Foo implements java.io.Serializable {
    int i1, i2;
    double d;
    int[] a;
    Object o;
    String s;
    Foo f;
}
```

Figure 6: An example serializable class: *Foo*.

can be connected. Properties that can be configured are for instance the serialization method that is used, reliability, message ordering, performance monitoring support, etc. This way, the layers on top of the IPL can configure the send and receive ports they create (and thus the channels between them) in a flexible way. Figure 5 shows an example code fragment that creates and configures a port type, and creates a send port of this new type.

3. IBIS IMPLEMENTATION

In this section, we will describe the Ibis implementation. An important part is the implementation of an efficient serialization mechanism. Although the Ibis serialization implementation was designed with efficient communication in mind, it is independent of the lower network layers, and completely written in Java. The communication code, however, has knowledge about the serialization implementation, because it has to know how the serialized data is stored to avoid copying. Applications can select at run time which serialization implementation (standard Sun serialization or optimized Ibis serialization) should be used for each individual communication channel. At this moment we have implemented two communication modules, one using TCP/IP and one using message passing (*MP*) primitives. The MP implementation can currently use Panda and MPI. The TCP/IP implementation is written in 100% pure Java, and runs on any JVM. The MP implementation requires some native code.

3.1 Efficient Serialization

Serialization is a mechanism for converting (graphs of) Java objects to a stream of bytes. Serialization can be used to ship objects between machines. One of the features of Java serialization is that the programmer simply lets the objects to be serialized implement the empty, special interface *java.io.Serializable*. Therefore, no special serialization code has to be written by the application programmer. The *Foo* class in Figure 6 is tagged as serializable in this way. The serialization mechanism always makes a deep copy of the objects that are serialized. For instance, when the first node of a linked list is serialized, the serialization mechanism traverses all references, and serializes the objects they point to (i.e., the whole list). The serialization mechanism can handle cycles, making it possible to convert arbitrary data structures to a stream of bytes. When objects are serialized, not only the object data is converted into bytes, but type and version information is also added. This way, when the stream is deserialized, the versions and types can be verified. When a version or type is unknown, the deserializer can use the bytecode loader to load the correct classfile for the type into the running application. Serialization performance is of critical importance for Ibis (and RMI), as it is used to transfer objects over the network (e.g., parameters to remote method invocations for RMI).

In previous work [19], we described what the performance bottlenecks in the serialization mechanism are, and how serialization can be made efficient when a native Java system is used (Manta

```
public final class FooGenerator extends Generator {
    public final Object doNew(ibis.io.IbisInputStream in)
      throws ibis.ipl.IbisIOException,
      ClassNotFoundException {
        return new Foo(in);
    }
}

class Foo implements java.io.Serializable,
                     ibis.io.Serializable {
    int i1, i2;
    double d;
    int[] a;
    String s;
    Object o;
    Foo f;

    public void ibisWrite(ibis.io.IbisOutputStream out)
      throws ibis.ipl.IbisIOException {
        out.writeInt(i1);
        out.writeInt(i2);
        out.writeDouble(d);
        out.writeObject(a);
        out.writeUTF(s);
        out.writeObject(o);
        out.writeObject(f);
    }

    public Foo(ibis.io.IbisInputStream in)
      throws ibis.ipl.IbisIOException,
      ClassNotFoundException {
        in.addObjectToCycleCheck(this);
        i1 = in.readInt();
        i2 = in.readInt();
        d = in.readDouble();
        a = (int[])in.readObject();
        s = in.readUTF();
        o = (Object)in.readObject();
        f = (Foo)in.readObject();
    }
}
```

Figure 7: Rewritten code for the *Foo* class.

in our case). The most important sources of overhead in standard Java serialization are run time type inspection, data copying and conversion, and object creation. Because Java lacks pointers, and information on object layout is not available, it is non-trivial to apply the techniques we implemented for Manta. We will now explain how we avoid these overhead sources in the Ibis serialization implementation.

The standard Java serialization implementation uses run time type inspection, called *reflection* in Java, to locate and access object fields that have to be converted to bytes. The run time reflection overhead can be avoided by generating serialization code for each class that can be serialized. This way, the cost of locating the fields that have to be serialized is pushed to compile time. Ibis provides a bytecode rewriter that adds generated serialization code to class files. This way, all programs can be rewritten, even when the source code is not available. The rewritten code for the Foo class from Figure 6 is shown in Figure 7 (we show Java code instead of bytecode for readability).

The bytecode rewriter adds a method that writes the object fields to a stream, and a constructor that reads the object fields from the stream into the newly created object. A constructor must be used for the reading side, because all *final* fields must be initialized in all constructors. Furthermore, the *Foo* class is tagged as rewritten with the same mechanism used by standard serialization: we let *Foo* implement the empty *ibis.io.Serializable* interface.

The generated write method (called *ibisWrite*) just writes the

```
public final void ibisWrite(ibis.io.IbisOutputStream out)
  throws ibis.ipl.IbisIOException {
    // Code to write fields i1 ... o is unchanged.
    boolean nonNull = out.writeKnownObjectHeader(f);
    if(nonNull) f.ibisWrite(out);
}

public Foo(ibis.io.IbisInputStream in)
  throws ibis.ipl.IbisIOException,
  ClassNotFoundException {
    // Code to read fields i1 ... o is unchanged.
    int i = in.readKnownTypeHeader();
    if(i == NORMAL_OBJECT) f = new Foo(in);
    else if(i == CYCLE) f = (Foo)in.getFromCycleCheck(i);
}
```

Figure 8: Optimization for *final* classes.

fields to the stream one at a time. The constructor that reads the data is only slightly more complicated. It starts with adding the *this* reference to the cycle check table, and continues reading the object fields from the stream that is provided as parameter. The actual handling of cycles is done inside the Ibis streaming classes. As can be seen in Figure 7, strings are treated in a special way. Instead of serializing the *String* object, the value of the string is directly written in the efficient UTF format.

The reading side has to rebuild the serialized object tree. However, in general, the exact class of the object to be created is unknown, due to inheritance. For example, the *o* field in the *Foo* object can refer to any non-primitive type. Therefore, type descriptors that describe the object's class have to be sent for each reference field. Using the type descriptor, an object of the actual type can be created. Because the type is not know at compile time, standard Java serialization uses *Class.newInstance* to create the required objects. We found that this operation is considerably more expensive than a normal *new* operation. For the IBM JIT, for example, *newInstance* takes about six times as long as a normal *new* operation.

Ibis implements an optimization that avoids the use of *newInstance*, making object creation cheaper. For each serializable class, a special *generator class* is generated (called *FooGenerator* in the example), which contains a method with a well-known name, *doNew*, that can be used to create a new instance of the accompanying serializable class (*Foo*). When a type is encountered for the first time, the *IbisInputStream* uses the expensive *newInstance* operation to create an object of the accompanying *generator* class. A reference to the generator object is then stored in a generator table in the *IbisInputStream*. When a previously encountered type descriptor is read again, the input stream can do a cheap lookup operation in the generator table to find the generator class for the type, and create a new instance of the desired class, by calling the *doNew* method on the generator. Thus, the *IbisInputStream* uses *newInstance* only for the first time a type is encountered. All subsequent times, a cheap table lookup is done, effectively eliminating a large part of the object creation overhead that is present in standard serialization.

The serialization mechanism can be further optimized when serializable classes are *final* (i.e., they cannot be extended). When a reference field that has to be serialized points to a final class, the type is known at compile time, and no type information has to be written to the stream. Instead, the deserializer can directly do a *new* operation for the actual class. Example code, generated by the bytecode rewriter, that implements this optimization, assuming that the *Foo* class is now final, is shown in Figure 8. The code only changes for the *f* field, because it has the (now final) type *Foo*. The other fields are omitted for brevity. The *ibisWrite* method can now directly call the *ibisWrite* method on the *f* field. The *writeKnownObjectHeader* method handles cycles and *null* references.

Some classes cannot easily be rewritten. for example, for some JVMs, it is impossible to rewrite the classes in the standard Java class libraries. Moreover, classes that were not rewritten can be received over the network. Therefore, Ibis serialization must handle classes that have not been rewritten. These classes can be recognized because they do not implement the *ibis.io.Serializable* interface, which is added by the bytecode rewriter when a class is rewritten. If an object was not rewritten, it is serialized using the normal (but slower) run time inspection techniques.

Ibis serialization tries to avoid the overhead of memory copies, as these have a relatively high overhead with fast networks, resulting in decreased throughputs. With the standard serialization method used by most RMI implementations, a zero-copy implementation is impossible to achieve, since data is always serialized into intermediate buffers. By using special typed buffers and treating arrays separately, Ibis serialization achieves zero-copy for arrays, and reduces the number of copies for complex data structures to one. The generated serialization code uses the *IbisInputStream* and *IbisOutputStream* classes to read and write the data. We will now explain these classes in more detail using Figure 9. The streaming classes use *typed buffers* to store the data in the stream for each primitive type separately. When objects are serialized, they are decomposed into primitive types, which are then written to a buffer of the same primitive type. No data is converted to bytes, as is done by the standard object streams used for serialization in Java. Arrays of primitive types are handled separately, and are stored in a separate array list. The stream data is stored in this special way to allow a zero copy implementation (which will be described in Section 3.2).

Figure 9 shows how an object of class *Bar* (containing two integers, a double and an integer array) is serialized. The *Bar* object and the array it points to are shown in the upper leftmost cloud labeled “application heap”. As the *Bar* object is written to the output stream, it is decomposed into primitive types, as shown in the lower cloud labeled “Ibis RTS heap”. The two integer fields *i1* and *i2* are stored in the *integer* buffer, while the double field *d* is stored in the separate *double* buffer. The array *a* is not copied into the typed buffers. Instead, a reference to the array is stored in a special array list. When the typed buffers are full, or when the *flush* operation is called on the stream, the data has is streamed.

For implementations in pure Java, all data has to be converted into bytes, because that is the only way to write data to TCP/IP sockets, UDP datagrams and files. Therefore, the typed buffer scheme is limited in its performance gain in a pure Java implementation. All types except floats and doubles can be converted to bytes using Java code. For floats and doubles, a native call inside the class libraries (*Float.floatToIntBits* and *Double.doubleToLongBits*) must be used for conversion to integers and longs, which can then be converted to bytes. When a float or double array is serialized, a native call is used for conversion *per element*. Because the Java native interface is quite expensive (see [16]), this is a major source of overhead for both standard and Ibis serialization. However, this overhead is unavoidable without the use of native code. Using the typed buffers mechanism and some native code, the conversion step from primitive types to bytes can be optimized by converting all typed buffers and all primitive arrays in the array list using *one* native call. For native implementations, conversion may not be needed; the typed buffer scheme then allows a zero copy implementation. On the receiving side, the typed buffers are recreated, as is shown in the right side of Figure 9. The read methods of the input stream return data from the typed buffers. When a primitive array is read, it is copied directly from the data stream into the destination array.

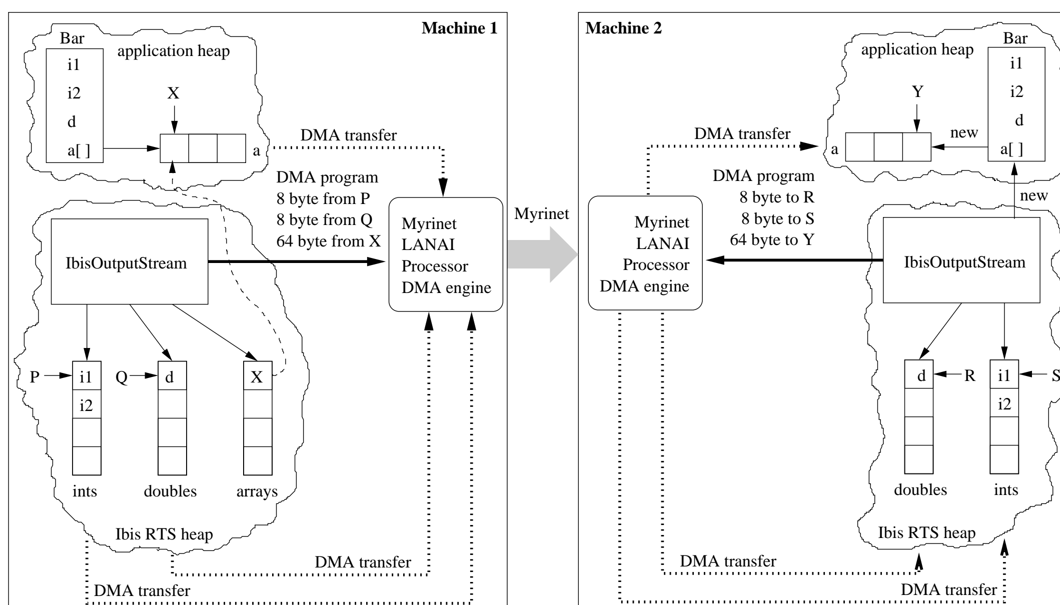

Figure 9: Low-level diagram of zero copy data transfer.

The rest of Figure 9 will be explained in the next section.

3.1.1 Ibis Serialization Performance

We ran several benchmarks to investigate the performance that can be achieved with the Ibis serialization scheme. Our measuring platform consists of 1 GHz Pentium III machines, connected by 100 Mbit Ethernet and Myrinet, running RedHat Linux with kernel 2.4.17. The results for different Java systems are shown in Table 1. The numbers show the performance of serialization to memory buffers, thus no communication is involved. The numbers show the host overhead caused by serialization, and give an upper limit on the communication performance. We show serialization and deserialization numbers separately, as they often take place on different machines, potentially in parallel. (i.e., when data is streamed). Two sets of numbers are shown for each JVM: the column labeled "conversion" includes the conversion to bytes, as is needed in a pure Java implementation. The column labeled "zero-copy" does not include this conversion and is representative for Ibis implementations that use native code. For communication bandwidth, deserialization is the limiting factor due to object creation and garbage collection. This is reflected in the table: the numbers for serialization (labeled "write") are generally higher than the numbers for deserialization (labeled "read").

We present numbers for arrays of primitive types and balanced binary trees with nodes that contain four integers. For the trees, we show both the throughput for the user payload (i.e., the 4 integers) and the throughput of the total data stream, including type descriptors and information that is needed to rebuild the tree (i.e., the references). The latter gives an indication of the protocol overhead. Some numbers are infinite, because zero-copy implementations just store references to arrays in the array list. The same holds for serialization of byte arrays, as these are not converted. The numbers show that the conversion of data is expensive, as the throughput numbers without the conversion (labeled zero-copy) are much higher. It is clear that, although no native code is used, high throughputs can be achieved with Ibis serialization, even for complex data structures.

3.2 Efficient Communication

It is well known that that in most (user level) communication systems most overhead is in software (e.g., data copying). Therefore, much can be gained by software optimization. In this section, we will describe the optimizations we implemented in the TCP/IP and Panda Ibis implementations. The general strategy that is followed in both implementations is to avoid thread creation, thread switching, locking, and data copying as much as possible.

3.2.1 TCP/IP Implementation

The TCP/IP Ibis implementation is relatively straightforward. One socket is used per unidirectional channel between a single send and receive port. However, we found that using a socket as a one-directional channel is inefficient. This is caused by the flow control mechanism of TCP/IP. Normally, acknowledgment messages are piggybacked on reply packets. When a socket is used in only one direction, there are no reply packets, and acknowledgments cannot be piggybacked. Only when a timeout has expired are the acknowledgments sent in a separate message. This severely limits the throughput that can be achieved. Because it is common that a runtime system (or application) creates both an outgoing and a return channel (for instance for RMI, see Figure 3), it is possible to optimized this scheme. Ibis implements channel pairing: whenever possible, the outgoing and return data channels are combined and use only one socket. This optimization greatly improves the throughput.

A socket is a one-to-one connection. Therefore, with multicast or many-to-one communication (e.g., multiple clients connecting to one server via RMI), multiple sockets must be created. A problem related to this is that Java does not provide a `select` operation. `select` can be used on a set of sockets, and blocks until data becomes available on any of the sockets in the set. Because Java does not offer a select operation, there are only two ways to implement many-to-one communication (both are supported by Ibis).

First, it is possible to poll a single socket, using the method *InputStream.available*. A set of sockets can thus be polled by just invoking *available* multiple times, once per socket. However, the

JVM	Sun 1.4				IBM 1.31				Manta			
	conversion		zero-copy		conversion		zero-copy		conversion		zero-copy	
	read	write	read	write	read	write	read	write	read	write	read	write
100 KB byte[]	92.4	∞	138.7	∞	108.5	∞	196.6	∞	134.0	∞	163.9	∞
100 KB int[]	78.9	220.0	136.7	∞	78.8	167.0	193.6	∞	57.8	162.8	133.8	∞
100 KB double[]	37.9	44.2	136.2	∞	48.4	42.4	194.1	∞	50.6	123.6	166.8	∞
1023 node tree user	32.0	47.4	40.4	79.7	42.1	51.4	60.6	71.3	22.0	38.1	32.5	59.9
1023 node tree total	48.4	71.5	61.0	120.4	63.6	77.7	91.5	107.8	33.1	57.4	48.9	90.1

Table 1: Ibis serialization throughput (MByte/s) for three different JVMs.

available method must do a system call to find out whether data is available for the socket. Hence, it is an expensive operation. Moreover, polling wastes CPU time. This is not a problem for single-threaded applications that issue explicit receive operations (e.g., MPI-style programs), but for multithreaded programs this is undesirable. When implicit receive is used in combination with polling, CPU time is always wasted, because one thread must be constantly polling the network to be able to generate upcalls.

Second, it is possible to use one thread per socket. Each thread calls a blocking receive primitive on the socket, and is unblocked by the kernel when data becomes available. This scheme does not waste CPU time, but now each thread uses memory space for its stack. Moreover, a thread switch is needed to deliver each message to the correct receiver thread when explicit receive is used. Ibis is flexible, and allows the programmer to decide which mechanism should be used via the properties mechanism described in Section 2.2.

3.2.2 Zero Copy Message Passing Implementation

The message passing (*MP*) implementation is built on top of native (written in C) message passing libraries, such as Panda and MPI. For each flush, the typed buffers and application arrays to be sent are handed as a message fragment to the MP device, which sends the data out without copying; this is a feature supported by both Panda and MPI. This is shown in more detail in Figure 9. On the receive side, the typed fields are received into pre-allocated buffers; no other copies need to be made. Only when sender and receiver have different byte order, a conversion pass is made over the buffers on the receiving side. Arrays are received in the same manner, with zero copy whenever the application allows it.

As with the TCP/IP implementation, multiplexing of Ibis channels over one device is challenging to implement efficiently. It is hard to wake up exactly the desired receiver thread when a message fragment arrives. For TCP/IP, there is support from the JVM and kernel that manage both sockets and threads: a thread that has done a receive call on a socket is woken up when a message arrives on that socket. A user-level MP implementation has a more difficult job, because the JVM gives no hooks to associate threads with communication channels. An Ibis implementation might use a straightforward, inefficient solution: a separate thread polls the MP device, and triggers a thread switch for each arrived fragment to the thread that posted the corresponding receive. We opted for an efficient implementation by applying heuristics to maximize the chance that the thread that pulls the fragment out of the MP device actually is the thread for which the fragment is intended.

A key observation is that a thread that performs a downcall receive is probably expecting to shortly receive a message (e.g., a reply). Such threads are allowed to poll the MP device in a tight loop for roughly two latencies (or until their message fragment arrives). After this polling interval, a yield call is performed to relinquish the CPU. A thread that performs an upcall service receives no such privileged treatment. Immediately after an unsuccessful poll, it yields the CPU to another polling thread.

JVM	Sun 1.4		IBM 1.31		Manta	
network	TCP	GM	TCP	GM	TCP	GM
latency downcall	143.2	61.5	133.8	33.3	146.8	35.1
latency upcall	128.6	61.5	126.0	34.4	140.4	37.0
Sun serialization						
100 KB byte[]	9.7	32.6	10.0	43.9	9.9	81.0
100 KB int[]	9.4	28.2	9.6	42.5	9.1	27.6
100 KB double[]	8.4	18.7	9.1	29.5	9.0	27.6
1023 node user	2.8	3.2	1.7	2.7	1.4	1.9
1023 node total	4.0	4.6	2.4	3.9	2.0	2.7
Ibis serialization						
100 KB byte[]	10.0	60.2	10.3	123	10.0	122
100 KB int[]	9.9	60.2	9.6	122	9.7	122
100 KB double[]	9.0	60.2	9.2	123	9.7	123
1023 node user	5.9	17.7	5.8	23.6	4.4	22.0
1023 node total	8.9	26.7	8.8	35.6	6.6	33.2

Table 2: Ibis communication latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet).

3.2.3 Performance Evaluation

Table 2 shows performance data for both Ibis implementations. For comparison, we show the same set of benchmarks as in Table 1 for both Sun serialization and Ibis serialization. Ibis is able to exploit the fast communication hardware and provides low communication latencies and high throughputs on Myrinet, especially when arrays of primitive types are transfered. The numbers show that Ibis provides portable performance, as all three Java systems achieve high throughputs with Ibis serialization. The array throughput on Myrinet (GM) for the Sun JIT is low compared to the other Java systems, because the Sun JIT makes a copy when a pointer to the array is requested in native code, while the other systems just use a pointer to the original object. Because this copying ruins the cache, throughput for the Sun JIT is better for messages of 40KB, where 83.4 MByte/s is achieved.

4. A CASE STUDY: EFFICIENT RMI

As described in Section 2.1, we implemented four runtime systems as a part of Ibis. In this paper, we focus on the RMI implementation, because RMI is present in standard Java, and many people are familiar with its programming model. The API of Ibis RMI is identical to Sun's RMI.

RMI is straightforward to implement, because most building blocks are present in the IPL. We extended the bytecode rewriter (which we use to generate serialization code) to generate the RMI stubs and skeletons. The RMI registry is implemented on top of the IPL registry. Communication channels are set up as shown in Figure 3. Thus, each stub has a send port, and each skeleton creates a receive port. When an RMI program issues a *bind* operation, the ports are connected. Using the properties mechanism described in Sec-

	Sun RMI	KaRMI 1.05b		Ibis RMI	
network	TCP	TCP	GM	TCP	GM
null-latency	218.3	127.9	32.2	131.3	42.2
array throughput					
100 KB byte[]	9.5	10.3	57.2	10.3	76.0
100 KB int[]	9.5	9.6	45.6	9.6	76.0
100 KB double[]	10.2	9.5	25.1	9.1	76.0
tree throughput					
1023 node user	2.2	2.3	2.5	4.3	22.9
1023 node total	3.2	3.3	3.6	6.5	34.6

Table 3: RMI latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet) using the IBM JIT.

tion 2.2, the ports can be configured to use Sun serialization or Ibis serialization.

Table 3 shows the latencies and throughputs that are achieved by several RMI implementations on our hardware using the IBM JIT. The Sun RMI implementation only runs over TCP. KaRMI [21] is an optimized serialization and RMI implementation. On TCP, KaRMI is implemented in pure Java, as is Ibis RMI on TCP. There also exists a version of KaRMI that uses native code to interface with GM, which makes KaRMI a good candidate for performance comparison with Ibis RMI, both on TCP and GM.

The throughput for sending `double` values with Sun RMI are higher than the throughputs achieved by KaRMI and Ibis RMI, because the IBM class libraries use a non-standard native method to convert entire double arrays to byte arrays, while the KaRMI and Ibis RMI implementations must convert the arrays using a native call per element. The latencies of KaRMI are slightly lower than the Ibis RMI latencies, but both are much better than the standard Sun RMI latency. Ibis RMI on TCP achieves similar throughput as KaRMI, but is more efficient for complex data structures, due to the generated serialization code. On Myrinet however, Ibis RMI outperforms KaRMI by a factor of 1.3 – 3.0 when arrays of primitive types are sent. For trees, the effectiveness of the generated serialization code and the typed buffer scheme becomes clear, and the Ibis RMI throughput is 9.1 times higher than KaRMI's throughput.

5. RELATED WORK

We discuss related work in four areas: grid computing, fast communication systems, parallel programming in Java and optimizations to serialization and RMI.

We have discussed a Java-centric approach to writing wide-area parallel (grid computing) applications. Most other grid computing systems (e.g., Globus [7] and Legion [11]) support a variety of languages. Converse [13] is a framework for multi-lingual interoperability. The SuperWeb [1], Javelin 2.0 [20], and Bayanihan [24] are examples of global computing infrastructures that support Java. A language-centric approach makes it easier to deal with heterogeneous systems, since the data types that are transferred over the networks are limited to the ones supported in the language (thus obviating the need for a separate interface definition language) [30].

Much research has been done since the 1980s on improving the performance of Remote Procedure Call protocols [12, 25]. Several important ideas resulted from this research, including the use of compiler-generated serialization routines and the need for efficient low-level communication mechanisms. Instead of kernel-level TCP/IP, Ibis can use efficient user-level communication substrates. Several projects are currently also studying protected user-level network access from Java, often using VIA [29].

Many other projects for parallel programming in Java exist. The JavaParty system [22] is designed to ease parallel cluster programming in Java. In particular, its goal is to run multithreaded programs with as little change as possible on a workstation cluster. JavaParty originally was implemented on top of Sun RMI, and thus suffered from the same performance problem as Sun RMI. The current implementation of JavaParty uses KaRMI.

An alternative for parallel programming in Java is to use MPI. Several MPI bindings for Java already exist [5, 9]. This approach has the advantage that many programmers are familiar with MPI. However, the MPI message-passing style of communication is difficult to integrate cleanly with Java's object-oriented model. MPI assumes an SPMD programming model that is quite different from Java's multithreading model. Also, current MPI implementations for Java suffer from the same performance problem as most RMI implementations: the high overhead of serialization and the Java Native Interface. For example, for the Java-MPI system described in [9], the latency for calling MPI from Java is 119 μs higher than calling MPI from C (346 versus 227 μs, measured on an SP2).

RMI performance is studied in several other papers. KaRMI is a new RMI and serialization package (drop-in replacement) designed to improve RMI performance [21]. The performance of Ibis RMI is better than that of KaRMI (see Table 3 in Section 4). The main reason is that Ibis generates serialization code instead of using run time type inspection. Also, Ibis exploits features of the underlying communication layer, and allows a zero-copy implementation by using typed buffers. Both KaRMI and Ibis RMI use a wire format that is different from the standard RMI format.

RMI performance is improved somewhat by Krishnaswamy et al. [15] by using caching and UDP instead of TCP. Their RMI implementation, however, still has high latencies (e.g., they report null-RMI latencies above a millisecond on Fast Ethernet). We found that almost all RMI overhead is in software, so only replacing the communication mechanism does not help much. Also, the implementation requires some modifications and extensions of the interfaces of the RMI framework. Sampemane et al. [23] describe how RMI can be run over Myrinet using the `socketFactory` facility.

6. CONCLUSIONS AND FUTURE WORK

Ibis allows highly efficient, object-based communication, combined with Java's "run everywhere" portability, making it ideally suited for high-performance computing in grids. The IPL provides a single, efficient communication mechanism using streaming and zero copy implementations. The mechanism is flexible, because it can be configured at run time using properties. Efficient serialization can be achieved by generating serialization code in Java, thus avoiding run time type inspection, and by using special, typed buffers to reduce type conversion and copying.

The Ibis strategy to achieving both performance and portability is to develop efficient solutions using standard techniques that work everywhere (for example, we generate efficient serialization code in Java), supplemented with highly optimized but non-standard solutions for increased performance in special cases. As test case, we studied an efficient RMI implementation that outperforms all previous implementations, without using native code. The RMI implementation also provides efficient communication on gigabit networks like Myrinet, but then some native code is required.

In future work, we intend to investigate adaptivity and malleability for the four programming models that are implemented as a part of Ibis, namely RMI, GMI, RepmI, and Satin. Ibis then provides dynamic information to the grid application about the available resources, including processors and networks. For this part, we are developing a tool called TopoMon [6], which integrates both topol-

ogy discovery and network monitoring. With the current implementation, Ibis enables Java as a quasi-ubiquitous platform for grid computing. In combination with the grid resource information, Ibis will leverage the full potential of dynamic grid resources to their applications.

Acknowledgments

This work is supported in part by a USF grant from the Vrije Universiteit, and by the European Commission, grant IST-2001-32133 (GridLab). We thank Ronald Veldema and Ceriel Jacobs for their work on the Manta system. Kees Verstoep and John Romein are keeping our hardware in good shape.

7. REFERENCES

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[3] F. Breg. *Java for High Performance Computing*. PhD thesis, University of Leiden, November 2001.

[4] J. Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. In *ACM 2001 Java Grande/ISCOPE Conference*, pages 97–105, June 2001.

[5] B. Carpenter, G. Fox, S. H. Ko, and S. Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. In *ACM 1999 Java Grande Conference*, pages 66–71, June 1999.

[6] M. den Burger, T. Kielmann, and H. E. Bal. TopoMon: A Monitoring Tool for Grid Network Topology. In *International Conference on Computational Science (ICCS 2002)*, April 2002.

[7] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, Summer 1997.

[8] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[9] V. Getov. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Supercomputing*, November 1999.

[10] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm Communications in Java for Grid Computing. *Communications of the ACM*, 44(10):118–125, 2001.

[11] A. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Comm. ACM*, 40(1):39–45, Jan. 1997.

[12] N. Hutchinson, L. Peterson, M. Abbott, and S. O'Malley. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proc. of the 12th ACM Symp. on Operating System Principles*, pages 91–101, Dec. 1989.

[13] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An interoperable framework for parallel programming. In *International Parallel Processing Symposium*, April 1996.

[14] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, May 1999.

[15] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient Implementations of Java RMI. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

[16] D. Kurzyniec and V. Sunderam. Efficient cooperation between Java and native codes – JNI performance benchmark. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001.

[17] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 2001.

[18] J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *LCR '02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, March 2002.

[19] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. on Programming Languages and Systems*, 23, 2001.

[20] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Euro-Par 2000 Parallel Processing*, number 1900, pages 1231–1238. Springer, Aug. 2000.

[21] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[22] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, Nov. 1997.

[23] G. Sampemane, L. Rivera, L. Zhang, and S. Krishnamurthy. HP-RMI : High Performance Java RMI over FM. University of Illinois at Urbana-Champaign, 1997.

[24] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5/6), 1999.

[25] C. Thekkath and H. Levy. Limits to Low-Latency Communication on High-Speed Networks. 11(2):179–203, May 1993.

[26] R. van Nieuwpoort, T. Kielmann, and H. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[27] R. V. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming using the Remote Method Invocation Model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[28] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July 1998.

[29] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

[30] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, May/June 1997.