Ibis: a Flexible and Efficient Java-based Grid Programming Environment

Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska,
Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Henri E. Bal
Faculty of Sciences, Department of Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

{rob,jason,gosia,rutger,ceriel,kielmann,bal}@cs.vu.nl

http://www.cs.vu.nl/ibis

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 [12]. 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, such as MPI [24], 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 [14]. 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 Just In Time compilers (JITs) 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 Java Virtual Machines (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 [32]. In earlier research on RMI [21], 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., using TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect. The *Ibis Portability Layer* (IPL) that provides this flexibility consists of a set of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. We will call a loaded implementation an *Ibis instantiation*.

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 implementations 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 this paper, we show that the IPL provides an extensible set of interfaces that allow the construction of dynamic grid applications in a portable way. The IPL communication primitives have been designed to allow efficient implementations. We demonstrate that the actual Ibis im-

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

plementation can be chosen at run time, by the application. Using generated serialization code, object streaming, and a zero-copy protocol, Ibis makes object-based communication efficient. We demonstrate the efficiency of Ibis with a fast RMI mechanism, implemented in pure Java. The Ibis approach of combining solutions that work "everywhere," combined with highly optimized solutions for special cases, provides efficiency and portability at the same time.

In Section 2 of this paper, we present the design of the Ibis system. Section 3 explains two different Ibis implementations. We present a case study of Ibis RMI in Section 4. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

2 Ibis Design

For deployment on the grid, it is imperative that Ibis is an extremely flexible system. Ibis should be able to provide communication support for any grid application, from the broadcasting of video to massively parallel computations. 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, like TCP/IP, UDP, GM. Moreover, Ibis should support malleability, i.e., machines must be able to join and leave a running computation.

Ibis consists of a runtime system (RTS) and a bytecode rewriter. The RTS implements the IPL. The bytecode rewriter is used to generate bytecode for application classes to actually use the IPL, a role similar to RMI's *rmic*. Below, we will describe how Ibis has been designed to support the aforementioned flexibility, while still achieving high performance.

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 JVM, but if a native, optimizing compiler (e.g., Manta [21]) 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, like GM or MPI, if available. Essentially, our aim is to reuse all good ideas from the Manta native Java system [21], but now implemented in pure Java. This is a non-trivial task, because pure Java lacks pointers, information on object layout, low level access to the thread package, 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 flexible enough to run seamlessly on a variety of different communication hardware and protocols;

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.

Thus, grid applications using Ibis can run on a variety of different machines, using optimized software (e.g., a native compiler, the GM Myrinet protocol, MPI, etc.) where possible, and using standard software (e.g., TCP) when necessary. Interoperability is achieved by using the well-known TCP protocol: when multiple Ibis implementations are used, e.g., one Ibis implementation on top of MPI, and one on top of GM, all machines can still be used in one single computation, using TCP across the different implementations. It is thus possible to simultaneously use GM for local communication and TCP for remote communication. There are currently two ways to achieve this with Ibis. A runtime system on top of the IPL can load two Ibis instantiations simultaneously, and choose the right one depending on the destination each message. Second, we provide an Ibis implementation that performs the bridging between multiple protocols internally. Some underlying communication systems may have a closed world assumption. In that case, Ibis also uses TCP to glue multiple "closed worlds" together. 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 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*. A software layer on top of the IPL can negotiate with Ibis instantiations 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 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. The IPL supports both reliable and unreliable communication, ordered and unordered messages, using a single, simple interface. Using user-definable properties (key-value pairs) 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 vice versa), and to create new objects on the receiving side, without using expensive reflection mechanisms. This way, the overhead of serialization is reduced dramatically.

Furthermore, our communication implementations use an optimized wire protocol. The Sun RMI protocol, for example, resends type information for each RMI. Our implementation 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, which are difficult to achieve with standard Java techniques. One example we investigated in previous work [21] is using a native, optimizing compiler instead of a JVM. This compiler (Manta), or any other high performance Java implementation, 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 Panda [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, 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 throughput. 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 possible for messages that transfer array data structures. For graph data structures, the number of copies can be reduced to one. Another major source of overhead is in the JNI (Java Native Interface) calls [20] required to convert floating point data types into bytes (and back). We found that, in combination with a zero-copy implementation, this problem can be solved by serializing into multiple buffers, one for each primitive data type. Implementing zero-copy (or single-copy) communication in Java is a non-trivial 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. The gray box denotes the Ibis system. An important component is the IPL or *Ibis Portability Layer*, which consists of a set of Java interfaces that define how the layers above Ibis 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.

Below the IPL are the modules that implement the actual Ibis functionality, such as serialization, communication, and typical grid computing requirements, such as performance monitoring and topology discovery. Although serialization and communication are mostly implemented inside Ibis, this is not required for all functionality. For many components, standard grid software can be used. Ibis then only contains an interface to these software packages.

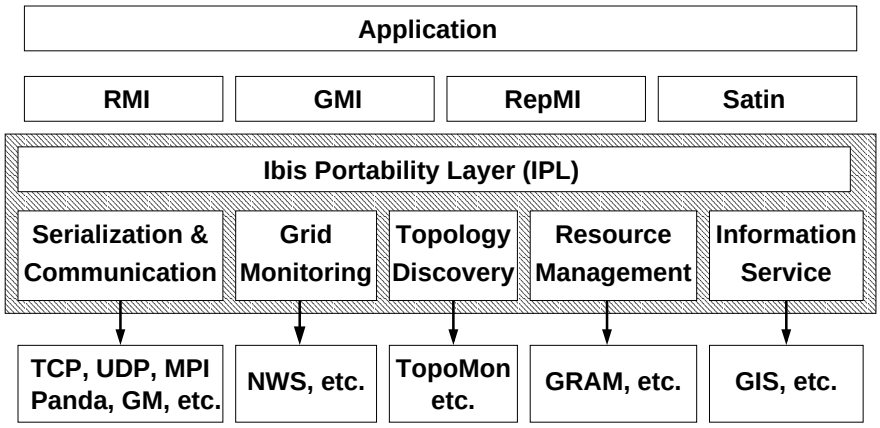

Figure 1: Design of Ibis. The various modules can be loaded dynamically, using run time class loading.

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 version 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 [33]. In [27] we demonstrated that parallel grid applications can be written with RMI. However, application-specific wide-area optimizations are needed.

GMI [23] extends RMI with group communication. 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 [18]), in order to make GMI more suitable for grid environments.

RepMI extends Java with efficient replicated objects. In [22] 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.

2.2 Design of the Ibis Portability Layer

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. The philosophy behind the design of the IPL is the following: whenever 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 taken.

2.2.1 Negotiating with Ibis Instantiations

Ibis implementations are loaded at run time. Ibis allows the application or runtime system on top of it to negotiate with the Ibis instantiations, in order to select the instantiation 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 instantiate 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.

Figure 2 shows example code that uses the IPL to load the best available Ibis implementation that meets the requirements of providing both FIFO ordering on messages and reliable communication. Although this code can be part of an application, it typically resides inside a runtime system for a programming model on top of the IPL. The *loadIbis* method is the starting point. First, it tries to load some user specified Ibis implementation, by calling *loadUserIbis*. If that fails, the code falls back to load the Panda implementation. If the Panda implementation cannot be loaded, the last resort is the TCP/IP implementation. The code in the *loadUserIbis* method shows how the layer above the IPL can negotiate with a loaded Ibis implementation about the desired properties. If the loaded Ibis implementation does not support the requirements, it can be unloaded. Subsequently, another implementation can be loaded and queried in the same way. Whether an implementation provides the desired properties has to be checked explicitly, because this can only be determined at runtime. The set of properties that is provided by an Ibis instantiation can depend on the availability of some native library or the version of the JVM that is used, for instance.

In contrast to many message passing systems, the IPL has no concept of hosts or threads, but uses location-independent *Ibis identifiers* to identify Ibis instantiations. A registry is provided to locate communication endpoints, called *send ports* and *receive ports*, using Ibis identifiers. The communication interface is object oriented. Applications using the IPL can create communication channels between objects (i.e. ports), 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.

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 to form a *unidirectional message channel*. The send port initiates the connection. Figure 3 shows such a channel. 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 *long* and *double*, and objects such as *String* or user-defined types can be written. When all data is inserted, the *send* primitive can be invoked on the message, sending it off.

```
Ibis loadUserIbis(String ibisImplName) {
    Ibis ibis = null;
    try {
        ibis = Ibis.createIbis("my ibis", ibisImplName);
    } catch (IbisException e) {
        return null; // Unable to load specified Ibis.
    }

    // An implementation is loaded, now see
    // whether it provides the properties we require.
    StaticProperties p = ibis.properties();

    // We need an Ibis that supports FIFO ordering.
    String ordering = p.find("message ordering");
    if(!ordering.equals("FIFO")) {
        ibis.unload();
        return null; // Not suported.
    }

    // We also need reliable communication.
    String reliability = p.find("reliability");
    if(!reliability.equals("true")) {
        ibis.unload();
        return null; // Not supported.
    }

    return ibis; // OK, return loaded implementation.
}

Ibis loadIbis(String ibisImplName) {
    Ibis ibis = loadUserIbis(ibisImplName);
    if(ibis == null) { // failed to load user Ibis
        try {
            ibis = Ibis.createIbis("my ibis",
                "ibis.ipl.impl.panda.PandaIbis");
        } catch (IbisException e) {
            // Could not load Panda/Ibis.
            try { // Fall back to TCP/Ibis.
                ibis = Ibis.createIbis(name,
                   "ibis.ipl.impl.tcp.TcpIbis");
            } catch (IbisException e) {
                return null; // All failed!
            }
        }
    }

    // We have loaded an Ibis implementation
    // with the desired properties.
    return ibis;
}
```

Figure 2: Loading and negotiating with Ibis implementations.

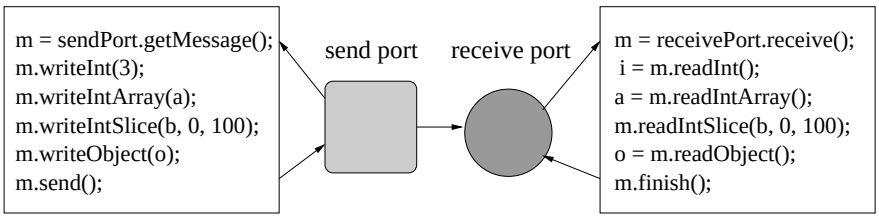

Figure 3: Send ports and receive ports.

The IPL offers two ways to receive messages. First, messages can be received with the receive port's blocking *receive* primitive (see Figure 3). 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, like RMI, GMI and Satin, rely on efficient implementations of upcalls. To avoid overheads of 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, like MPI or Panda, are connectionless. Messages are sent to their destination, without explicitly creating a connection first. In contrast, the IPL 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.

The standard Java streaming classes (used for serialization and for writing to TCP/IP sockets) convert all data structures to bytes, including the primitive types and arrays of primitive types. An exception is only made for byte arrays. Furthermore, there is no support for slices of arrays. When a pure Java implementation is used, the copying of the data is thus unavoidable.

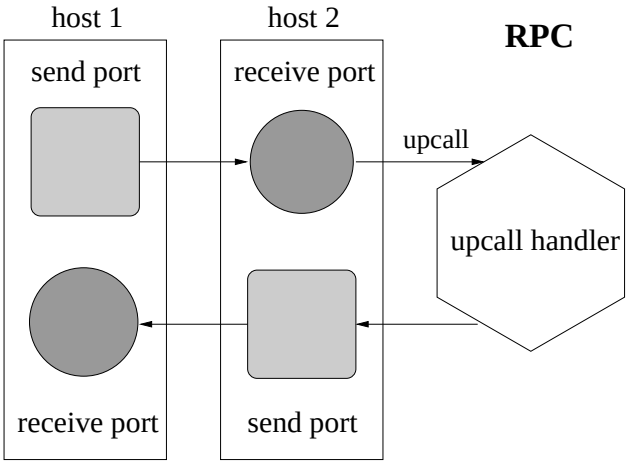

Figure 4: An RPC implemented with the IPL primitives.

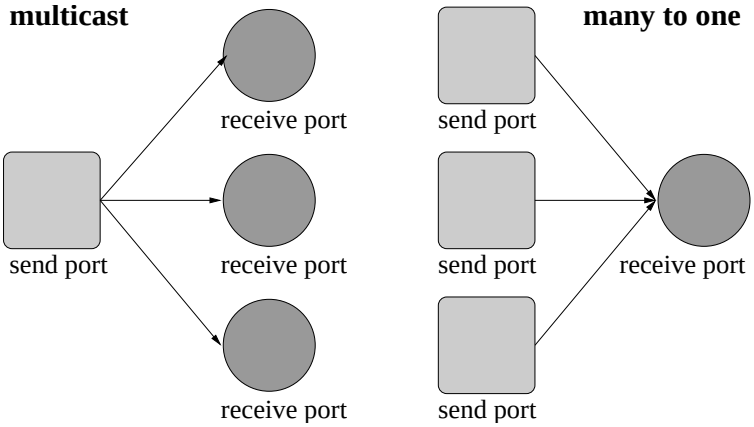

Figure 5: Other IPL communication patterns.

An important insight is that zero-copy can be made possible in some important special cases by carefully designing the interface for reading data from and writing data to messages. As can be seen in Figure 3, the IPL provides special read and write methods for (slices of) all primitive arrays. This way, Ibis allows 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 provided by the IPL. 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 Figure 4 shows. 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 5). Furthermore, multiple send ports can connect to a single receive port, thus implementing many-to-one communication, also shown in Figure 5.

feature / system	UDP	TCP	Panda	MPI	Nexus	IPL
FIFO ordering	no	yes	yes	yes	yes	yes
unordered messages	yes	no	no	yes	yes	yes
reliable messages	no	yes	yes	yes	yes	yes
unreliable messages	yes	no	no	no	yes	yes
streaming data	no	yes	no	no	no	yes
unordered multicast	no	no	yes	yes	yes	yes
totally ordered multicast	no	no	yes	yes	yes	yes
malleability	yes	yes	no	no	yes	yes
multithreading support	no	no	yes	no	yes	yes
upcalls	no	no	yes	no	yes	yes
asynchronous upcalls	no	no	yes	no	no	yes
explicit receive	yes	yes	no	yes	yes	yes
complex data structures	no	no	yes	yes	no	yes
dynamic/linked data structures	no	no	yes	no	no	yes
multiple protocols	no	no	no	no	yes	yes
connection oriented	no	yes	no	no	yes	yes
multiple language bindings	yes	yes	no	yes	no	no

Table 1: Features of communication systems.

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 with properties (key-value pairs) via the IPL. Only ports of the same type 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.

2.3 Comparison with Other Communication Systems

The extreme flexibility of the IPL is shown by the features that are supported in the interface. Table 1 lists several features of communication systems, and compares the IPL with UDP, TCP, Panda [2], MPI [24], and Nexus [8]. The great strength of the IPL is that all features can be accessed via one single interface. The layers above the IPL can enable and disable features as wanted using key-value pairs (properties). Moreover, the IPL is extensible, as more properties can be added without changing the interface.

TCP and UDP are widely used communication protocols that are also supported in Java. However, the programming interfaces (i.e., sockets and unreliable datagrams) offered by the protocols are too low level for use in a parallel programming context. TCP does not provide multicast, although this is important for many parallel applications. UDP does provide a multicast mechanism, but it provides only unreliable communication, and no ordering is guaranteed between messages. Parallel programs often require both properties, however.

Panda is an efficient communication substrate that was designed for runtime systems for parallel languages. Panda is not suitable for grid environments because it does not support malleability. Panda is described in more detail in [2].

MPI [24] (Message Passing Interface) is a message passing library that is widely used for parallel programming. However, the programming model of MPI is designed for SPMD style computations, and lacks support for upcalls and multithreading, which are essential for efficient RPC and RMI implementations. Moreover, although MPI supports complex data structures, it cannot handle dynamic data structures like linked lists and hash tables. MPI was initially targeted for shared memory machines and clusters of workstations, Recently, however, there has been a large interest in using MPI in grid environments [7, 9, 10, 18]. 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.

Nexus [8] is a communication library that was used in the early versions Globus toolkit. Although Nexus is no longer supported, the way Nexus deals with multiple protocols is interesting. Nexus, like Ibis, supports automatic selection of optimal protocols at run time. However, Nexus chooses the protocol based on the hardware it runs on, while Ibis is more flexible: it allows the layer on top of the IPL to negotiate about the protocol that should be used. This is important, because the optimal protocol may depend on the application, not only on the hardware that is available.

Nexus is written in C, and is therefore portable only in the traditional sense. The interface provided by Nexus is similar to the IPL: the only supported mechanism is a connection oriented one way channel that allows asynchronous messages. However, Nexus does not support asynchronous upcalls (i.e., polling is required to trigger upcalls), and more importantly, Nexus does not provide a streaming mechanism, as a single data buffer is passed to the asynchronous message when it is sent. We found that the streaming of data is imperative to achieve good performance with complex data structures. The performance results provided in [8] show that Nexus adds considerable overhead to the low-level communication mechanisms that it is built upon.

3 Ibis Implementations

In this section, we will describe the Ibis implementations. 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, in order 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 either Panda or MPI. The TCP/IP implementation is written in 100% pure Java, and runs on any JVM. The MP implementation requires some native code to convert JNI calls to native Panda or MPI calls.

```
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*.

3.1 Efficient Serialization

Serialization is a mechanism for converting (graphs of) Java objects to some format that can be stored or transfered (e.g., a stream of bytes, or XML). 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. As an example, 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 to bytes, but type and version information is also added. When the stream is deserialized, the versions and types are examined, and objects of the necessary type can be created. 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, such as parameters to remote method invocations.

In previous work [21], we identified the performance bottlenecks in the serialization mechanism, and how it can be made efficient when a native Java system like Manta is used. The most important sources of overhead in standard Java serialization are run-time type inspection, data copying and conversion, object creation and the use of an inefficient protocol. Because Java lacks pointers, and information on object layout is not available either, it is non-trivial to apply the techniques we implemented for Manta. We will now explain how we avoid these sources of overhead in the Ibis serialization implementation.

3.1.1 Avoiding Run Time Type Inspection

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 moved from run time to compile time. Ibis provides a bytecode rewriter that adds generated serialization code to class files, allowing all programs to be rewritten, even when the source code is not available. The rewritten code for

```
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.

the *Foo* class from Figure 6 is shown in Figure 7. There, 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 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,

```
int typeDescriptor = readInt();

Generator gen = getFromGeneratorTable(typeDescriptor);
if (gen == null) { // Not seen before.
    String name = readClassName();
    gen = Class.newInstance(name + "Generator");
    addToGeneratorTable(typeDescriptor, gen);
}

return gen.doNew();
```

Figure 8: Pseudo code for optimized object creation.

the value of the string is directly written in the more efficient UTF format.

3.1.2 Optimizing Object Creation

The reading side has to rebuild the serialized object graph. In general, the exact type of objects that have to be created is unknown, due to inheritance. For example, the *o* field in the *Foo* object from Figure 7 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. Standard Java serialization uses a *private* native method inside the standard class libraries to create objects without invoking a constructor, as a constructor may have side-effects. We found that this is considerably more expensive than a normal *new* operation.

Ibis serialization implements an optimization that avoids the use of this native method, making object creation cheaper. For each serializable class, a special *generator class* (called *FooGenerator* in Figure 7) is generated by the bytecode rewriter. The generator class contains a method with a well-known name, *doNew*, that can be used to create a new object of the accompanying serializable class, like *Foo*.

Pseudo code that implements the object-creation optimization using the generator class is shown in Figure 8. When a type is encountered for the first time, the *IbisInputStream* uses the standard, but expensive *Class.newInstance* operation to create an object of the accompanying *generator* class. A reference to the generator object is then stored in a 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 object 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 and a normal *new* is done instead of the call to a native method that is used by standard Java serialization. Side effects of user defined constructors are avoided because *doNew* calls the *generated* constructor that reads the object fields from the input stream. A user defined constructor is never invoked. This optimization effectively eliminates 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 via inheritance). 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

```
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);
    else f = null;
}
```

Figure 9: Optimization for *final* classes.

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[]	115.9	∞	134.2	∞	121.6	∞	120.4	∞	187.8	∞	195.3	∞
100 KB int[]	77.6	282.3	118.4	∞	82.4	186.4	110.4	∞	68.8	195.3	160.1	∞
100 KB double[]	36.2	43.3	135.1	∞	49.9	45.2	121.2	∞	54.3	111.6	203.5	∞
1023 node tree user	41.7	50.6	61.0	71.0	31.7	49.0	38.6	80.5	23.0	35.5	36.5	60.0
1023 node tree total	63.0	76.4	92.2	107.3	47.9	74.0	58.3	121.7	34.7	53.3	55.2	90.8

Table 2: Ibis serialization throughput (MByte/s) for three different JVMs.

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 9. 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. On the reading side, a normal *new* operation can be done, as the type of the object that has to be created is known at compile time.

Some classes cannot easily be rewritten. For example, classes that were received over the network. These classes can be recognized because they do not implement the *ibis.io.Serializable* interface (which is added by the Ibis bytecode rewriter). If a class was not rewritten, objects of that class are serialized using the normal (but slower) run time inspection techniques.

3.1.3 Avoiding Data Copying

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 classes *IbisIn-*

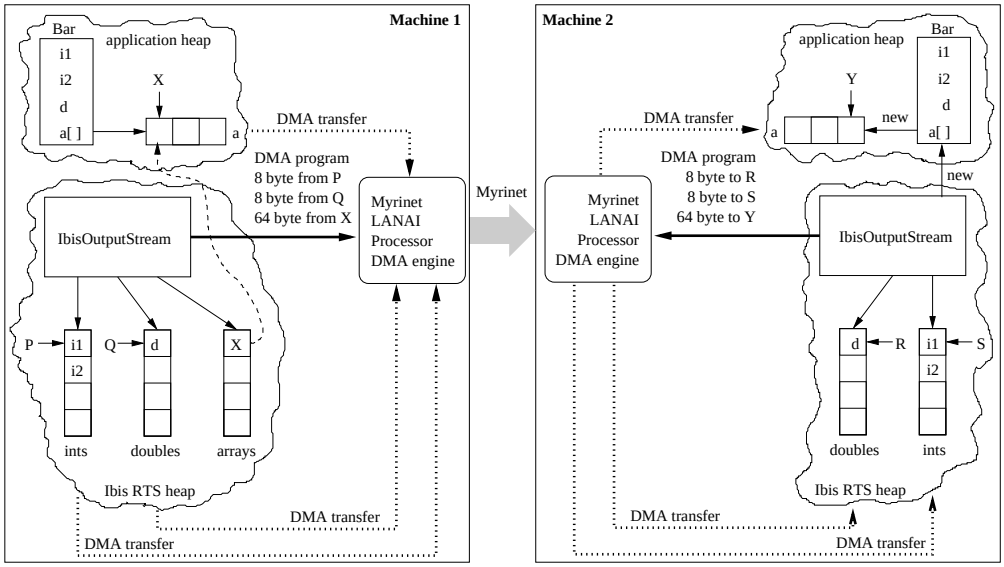

Figure 10: Low-level diagram of zero copy data transfers with the Ibis implementation on Myrinet.

putStream and *IbisOutputStream* to read and write the data. We will now explain these classes in more detail, using Figure 10. 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 specially: a reference to the array is stored in a separate array list, no copy is made. The stream data is stored in this special way to allow a zero copy implementation (which will be described in Section 3.2).

Figure 10 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 invoked, the data is streamed.

For implementations in pure Java, all data has to be converted to 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, Java provides a native method inside the class libraries (*Float.floatToIntBits* and *Double.doubleToLongBits*), that must be used for conversion to integers and longs, which can then be converted to bytes. Because these native methods must be provided by all JVMs, their use does not interfere with Ibis' portability requirements. 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 [20], this is a major source of overhead

JVM	Sun 1.4		IBM 1.31		Manta	
	read	write	read	write	read	write
100 KB byte[]	99.6	∞	97.0	∞	171.3	∞
100 KB int[]	64.3	151.0	68.2	144.9	87.2	113.6
100 KB double[]	32.7	38.4	54.8	61.0	85.7	120.6
1023 node tree user	8.2	12.4	3.3	6.0	5.9	15.6
1023 node tree total	11.8	17.9	4.7	8.6	8.6	22.4

Table 3: Standard Java serialization throughput (MByte/s) for three different JVMs.

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 communication 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 10. 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. The parts of Figure 10 that have not yet been discussed will be explained in the next section.

3.1.4 Ibis Serialization Performance

We have run 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 both by 100 Mbit Ethernet and by Myrinet, running RedHat Linux 7.2 with kernel 2.4.9-13. The results for Ibis serialization on different Java systems are shown in Table 2, while the performance of standard Java serialization is shown in Table 3 for reference. The figures give 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. Due to object creation and garbage collection, deserialization is the limiting factor for communication bandwidth. This is reflected in the table: the numbers for serialization (labeled "write") are generally higher than the numbers for deserialization (labeled "read").

For Ibis serialization, 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 to implement zero-copy communication.

We present numbers for arrays of primitive types and balanced binary trees with nodes that contain four integers each. For the trees, we show both the throughput for the user payload (i.e., the 4 integers) and 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 throughput numbers are infinite, indicated by the ∞-signs in the table, because zero-copy implementations can have constant overhead in cases where they just store references to arrays in the array list, which is independent of the amount of data being transfered. The same holds for serialization of byte arrays, as these are not converted.

The numbers show that data conversion is expensive: throughputs without 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, especially for complex data structures. For reading binary trees, for instance, Ibis serialization with data conversion achieves a payload throughput that is 5.0 times higher than the throughput of standard serialization on the Sun JIT, and 9.6 times higher on the IBM JIT. When data conversion is not needed, the difference is even larger: the payload throughput for Ibis serialization is 7.4 times higher on the Sun JIT and 11.6 times higher on the IBM JIT.[3] On Manta, the throughput for arrays is better for Sun than for Ibis serialization, because Manta avoids clearing new objects (initializing with zero values) with its native Sun serialization implementation.

3.2 Efficient Communication

It is well known that in (user level) communication systems most overhead is in software, e.g., data copying. Therefore, much can be gained from software optimization. In this section, we will describe the optimizations we have 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 one direction only, there are no reply packets, and acknowledgments cannot be piggybacked. Only when a timeout has expired, the acknowledgments are sent in separate messages. This severely limits the throughput that can be achieved. Because it is common that a run-time system of application creates both an outgoing and a return channel (for instance for RMI, see Figure 4), it is possible to optimize this scheme. Ibis implements channel pairing: whenever possible, the outgoing and return data channels are combined, using 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, like multiple clients connecting to one server via RMI, multiple sockets must be created. A related problem is that not all Java implementations provide a *select* primitive[4]. The *select* operation can be used on a set of sockets, and blocks until data becomes available on any of the sockets in the set. A *select* operation cannot be implemented in native code, because Java does not export file descriptors. We must also provide a solution when *select* is not present. In that case, there are only two ways to implement many-to-one communication (both are supported by Ibis).

[3] Separate measurements show that using the *java.io.Externalizable* interface can improve the performance of SUN serialization slightly. However, Ibis serialization still is significantly faster for all tests. Moreover, when *java.io.Externalizable* is used, the user has to write all serialization code by hand.

[4] A *select* operation has recently been added in Java 1.4, in the *java.nio* package. Because we use Ibis for grid computing, and the latest features of Java may not be available on all platforms, we must support older Java versions.

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 *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, 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 messages to the actual receiver thread when explicit receive is used. Ibis 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 10. On the receiving 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 single 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.

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 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. A straightforward, inefficient solution is to let a separate thread poll the MP device, and trigger 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 message latencies, or until their message 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 thread.

3.2.3 Performance Evaluation

Table 4 shows performance data for both Ibis implementations. For comparison, we show the same set of benchmarks as in Table 2 for both Sun serialization and Ibis serialization. Ibis is able

JVM	Sun 1.4		IBM 1.31		Manta	
network	TCP	GM	TCP	GM	TCP	GM
latency downcall	143	61.5	134	33.3	147	35.1
latency upcall	129	61.5	126	34.4	140	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 tree user	2.8	3.2	1.7	2.7	1.4	1.9
1023 node tree 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 tree user	5.9	17.7	5.8	23.6	4.4	22.0
1023 node tree total	8.9	26.7	8.8	35.6	6.6	33.2

Table 4: Ibis communication latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet).

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 affects the data cache, throughput for the Sun JIT is better for smaller messages of 40KB, where 83.4 MByte/s is achieved.

4 A case study: Efficient RMI

As described in Section 2.1, we have implemented four runtime systems as part of Ibis. In this paper, we elaborate 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. We briefly describe the RMI implementation on top of the IPL and investigate the performance with microbenchmarks and an application. We compare the Ibis RMI performance with Sun RMI, KaRMI [28], and with C and MPI.

4.1 RMI Implementation

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

	C / MPI		IPL		Sun RMI	KaRMI 1.05b		Ibis RMI	
network	UDP	GM	TCP	GM	TCP	TCP	GM	TCP	GM
null-latency	161	22	126	34.4	218.3	127.9	32.2	131.3	42.2
array throughput									
100 KB byte[]	11.1	143	10.3	123	9.5	10.3	57.2	10.3	76.0
100 KB int[]	11.1	143	9.6	122	9.5	9.6	45.6	9.6	76.0
100 KB double[]	11.1	143	9.2	123	10.2	9.5	25.1	9.1	76.0
tree throughput									
1023 node tree user	5.4	16.6	5.8	23.6	2.2	2.3	2.5	4.3	22.9
1023 node tree total	8.0	30.0	8.8	35.6	3.2	3.3	3.6	6.5	34.6

Table 5: Latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet) using C / MPI and the IBM JIT.

4.2 RMI Performance

Table 5 shows the latencies and throughputs that are achieved by several RMI implementations on our hardware using the IBM JIT. For comparison, we also provide numbers for C / MPI and IPL level communication. For Fast Ethernet, we use MPICH-P4, for Myrinet we use MPICH-GM. The Sun RMI implementation only runs over TCP. KaRMI [28] is an optimized serialization and RMI implementation. On TCP, KaRMI is implemented in pure Java, as is Ibis RMI on TCP. There also is a KaRMI version 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 Ibis performance at the IPL level is close to the MPI performance. The IPL throughput on Myrinet, for instance, is only 14% lower than the MPI throughput. For complex data structures Ibis even outperforms MPI, even though the MPI tree benchmark uses cached send and receive buffers that are large enough to contain the whole tree.

As can be seen, RMI has a substantial performance overhead on Myrinet compared to IPL level communication. The higher latency is mainly caused by thread creation and thread switching at the receiving side, which are a result of the RMI model: RMI uses implicit message receipt (popup or upcall threads). Also, an ”empty” RMI call still transmits an object identifier and a method, and requires the object and method to be looked up in a table at the receiver, while an empty IPL level message requires no data or processing at all. The lower RMI array throughput on Myrinet is caused by the fact that the JVM must allocate a new array for each incoming invocation, and also zeroes (clears) this newly allocated array. Moreover, the garbage collector has to be activated to reclaim the arrays. Using the low-level Ibis communication primitives, the same array can be reused for each incoming message, thus avoiding memory management overhead and also resulting in better caching behavior.

The throughput on TCP for sending *double* values with Sun RMI are higher than the throughputs achieved by KaRMI and Ibis RMI, because the IBM class libraries internally 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.

4.3 Ibis RMI Application Performance

To investigate the performance of Ibis RMI at the application level, we implemented Successive Over-Relaxation, in C with MPI, in Java using the IPL for communication, and in Java with RMI. The idea is that for Java and Ibis to be useful for high-performance grid computing, its application level performance should be close to that of traditional C / MPI applications. SOR is an iterative algorithm for solving Laplace equations on a grid data structure. The sequential algorithm works as follows. For all non-boundary points of the grid, SOR first calculates the average value of the four neighbors of the point:

$$av(r,c) = \frac{g[r+1,c+1] + g[r+1,c-1] + g[r-1,c+1] + g[r-1,c-1]}{4}$$

Then the new value of the point is determined using the following correction:

$$g_{new}[r,c] = g[r,c] + \omega(av(r,c) - g[r,c])$$

ω is known as the *relaxation parameter* and a suitable value can be calculated in the following way:

$$\omega = \frac{2}{1 + \sqrt{1 - \frac{1}{4}(\cos\frac{\pi}{totalcolumns} + \cos\frac{\pi}{totalrows})^2}}$$

The entire process terminates if during the last iteration no point in the grid has changed more than a certain (small) quantity *maxDiff*. The parallel implementation of SOR is based on the Red/Black SOR algorithm. The grid is treated as a checkerboard and each iteration is split into two phases, red and black. During the red phase only the red points of the grid are updated. Red points only have black neighbors and no black points are changed during the red phase. During the black phase, the black points are updated in a similar way. Using the Red/Black SOR algorithm the grid can be partitioned among the available processors, each processor receiving a number of adjacent rows. All processors can now update different points of the same color in parallel. Before a processor starts the update of a certain color, it exchanges the border points of the opposite color with its neighbor. After each iteration, the processors must decide if the calculation must continue. An *all-reduce* operation is used to determine the maximum change of the grid points. If the largest change is smaller than *maxDiff*, the program terminates. In short, SOR uses neighbor communication to exchange the grid rows, and a collective reduce operation to test the stop condition. Both operations are performed each iteration.

For the C measurements, we use the GNU C compiler version 3.3.2, with the "-O3" flag to turn optimizations on. For Java, we use the IBM JIT version 1.3.1, because it is significantly faster than the SUN JIT. We use a grid of 8192×8192, which occupies 512 MByte of memory. With this problem size, SOR needs 151 iterations to complete. The run time for the sequential C version is 1407.4 seconds, while the Java version runs for 1466.6 seconds. The overhead of using Java instead of C is thus only 4%.

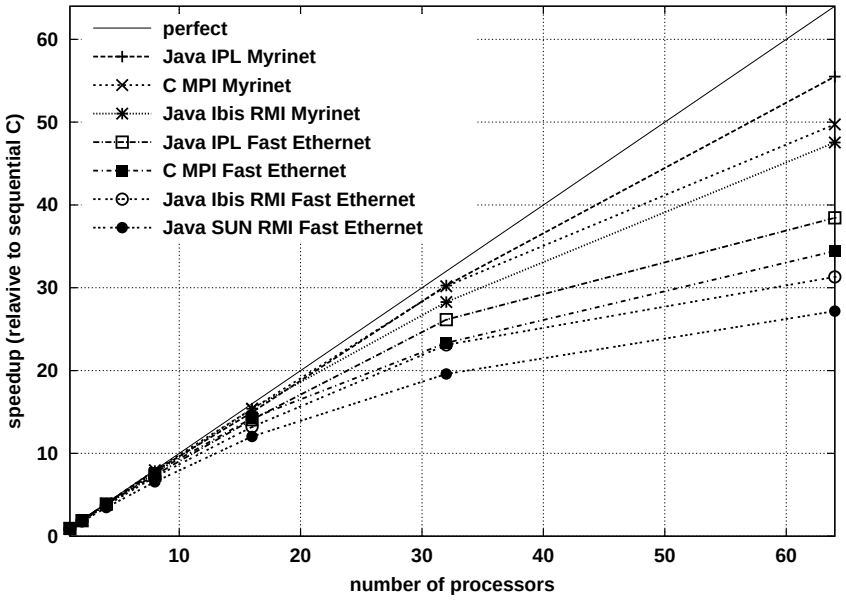

Figure 11: Performance of the SOR application on Fast Ethernet and Myrinet, with C / MPI, Ibis IPL, Ibis RMI and Sun RMI.

Figure 11 shows the speedups achieved by the parallel versions of SOR. The graph shows the speedups of the C / MPI version, a Java version written directly on top of the IPL, and a Java version written with RMI. We ran the RMI program both with Sun RMI and Ibis RMI. All speedups, also for the Java versions, are computed relative to the sequential C program. Therefore, it is possible to directly compare the lines in the graph: a higher line means a faster execution time. Although the sequential C version is slightly faster than the sequential Java program, the parallel Java SOR implemented directly on top of the IPL scales better, and actually *outperforms* the C / MPI version on a larger number of machines, both on Fast Ethernet and Myrinet.

Using the Java RMI model instead of the IPL causes a noticeable performance drop. The microbenchmarks in Section 4.2 show that the overhead of RMI is significant, and the application measurements confirm this. Still, the performance difference between Ibis RMI and C / MPI on 64 machines is only 10% on Fast Ethernet, and less than 5% on Myrinet. The graph makes clear that Sun RMI is significantly slower than Ibis RMI at the application level.

5 Related Work

We have discussed a Java-centric approach to writing wide-area parallel (grid computing) applications. Most other grid computing systems (e.g., Globus [11] and Legion [15]) support a variety of languages. GridLab [31] is building a toolkit of grid services that can be accessed from various programming languages. Converse [16] is a framework for multi-lingual interoperability. The SuperWeb [1], Javelin 2.0 [25], and Bayanihan [30] 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) [35]. Recently, web services [12] have become the de-facto standard for communication between grid services. Although the performance of web services is adequate for communication between grid services, for parallel programming, high performance communication is critical. For web services to support high performance distributed scientific computing, the standard must be modified or extended [17] to achieve acceptable performance.

Much research has been done since the 1980s on improving the performance of Remote Procedure Call protocols [32]. Several important ideas resulted from this research, including the use of compiler-generated serialization routines and the need for efficient low-level communication mechanisms. Ibis can use efficient user-level communication substrates instead of kernel-level TCP/IP. Several projects are currently also studying protected, user-level network access from Java, often using VIA [34]. Our communication substrate Panda [2] also uses one interface for different communication hardware. However, with Panda, this selection is performed at compile time; Ibis can switch implementations at run time. A communication library in the Globus toolkit, Nexus [8], also supports automatic selection of optimal protocols at run time.

Many other projects for parallel programming in Java exist. Titanium [36] is a Java-based language for high-performance parallel scientific computing. It extends Java with features like immutable classes, fast multidimensional array access, and an explicitly parallel SPMD model of communication. The JavaParty system [29] 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 pure Java is to use MPI. Several MPI bindings for Java already exist [5, 13]. 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 in [13], 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 an improved RMI and serialization package designed to improve RMI performance [28]. The performance of Ibis RMI is better than that of KaRMI (see Table 5 in Section 4.2). 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. RMI performance is improved somewhat by Krishnaswamy et al. [19] 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.

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 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 programming models that are implemented in Ibis (i.e., 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 topology 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 for his work on Manta. Kees Verstoep and John Romein are keeping our hardware in good shape.

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 Trans. on Computer Systems*, 16(1):1–40, 1998.

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

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

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

[6] Mathijs den Burger, Thilo Kielmann, and Henri E. Bal. TopoMon: A Monitoring Tool for Grid Network Topology. In *Intl. Conf. on Computational Science*, volume 2330 of *LNCS*, pages 558–567, 2002.

[7] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, 1998.

[8] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high- performance networked computing systems. *J. Parallel and Distributed Computing*, 40:35–48, 1997.

[9] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the Message Passing. In *MPI Developer Conference*, 1996.

[10] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'98)*, Orlando, FL, November 1998. ACM Press, Online at http://www.supercomp.org/sc98/proceedings/.

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

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

[13] V. Getov. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*, 1999. ACM Press, Online at http://www.supercomp.org/sc99/proceedings/.

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

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

[16] Laxmikant V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An interoperable framework for parallel programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[17] Randall Bramley Kenneth Chiu, Madhusudhan Govindaraju. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of The Eleventh International Symposium on High Performance Distributed Computing*, pages 246–254, Edinburgh, Scotland, July 2002. IEEE Computer Society Press.

[18] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul 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*, pages 131–140, 1999.

[19] 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'98)*, pages 19–36, Santa Fe, NM, 1998.

[20] Dawid Kurzyniec and Vaidy Sunderam. Efficient cooperation between Java and native codes – JNI performance benchmark. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPDA)*, Las Vegas, Nevada, USA, June 2001.

[21] J. Maassen, Rob V. van Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. on Programming Languages and Systems*, 23(6):747–775, 2001.

[22] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Parallel application experience with replicated method invocation. *Concurrency & Computation: Practice and Experience*, 13(8-9):681–712, 2001.

[23] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 2002. To be published in Lecture Notes in Computer Science, Springer-Verlag.

[24] MPI Forum. MPI: A Message Passing Interface Standard. *Int. J. Supercomputer Applications*, 8(3/4):157–416, 1994.

[25] Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Euro-Par 2000 Parallel Processing*, number 1900 in LNCS, pages 1231–1238, 2000.

[26] Rob V. van Nieuwpoort, T. Kielmann, and H.E. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, 2001.

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

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

[29] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[30] Luis F. G. Sarmenta. *Volunteer Computing*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2001.

[31] Ed Seidel, Gabrielle Allen, André Merzky, and Jarek Nabrzyski. GridLab -a Grid Application Toolkit and Testbed. *Future Generation Computer Systems*, 18, 2002.

[32] C.A. Thekkath and H.M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Trans. on Computer Systems*, 11(2):179–203, May 1993.

[33] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July 1998.

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

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

[36] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-performance Java Dialect. In *ACM 1998 workshop on Java for High-performance network computing*, 1998.