Efficient Remote Method Invocation

Ronald Veldema Rob van Nieuwpoort Jason Maassen
Henri E. Bal Aske Plaat[*]

Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
rveldema@cs.vu.nl rob@cs.vu.nl jason@cs.vu.nl bal@cs.vu.nl aske@cs.vu.nl
`http://www.cs.vu.nl/albatross/`

Technical Report IR-450

September 14, 1998

Abstract

In current Java implementations, Remote Method Invocation is slow. On a Pentium Pro/Myrinet cluster, for example, a null RMI takes 1228 μs using Sun's JDK 1.1.4. This paper describes Manta, a Java system designed to support efficient communication. On the same Myrinet cluster, Manta achieves a null RMI latency of 35 μs. Manta is based on a native Java compiler. It achieves high communication performance mainly through the following techniques: removal of copying in the network sub system, compiler-generation of specialized marshaling code, and Optimistic Active Messages, a technique that avoids thread creation in message handlers for small methods. Our work shows that techniques from other high performance RPC based systems also work in Java, making RMI, originally designed for distributed programming, a viable mechanism for parallel programming as well.

1 Introduction

There is a growing interest in using Java for high-performance parallel applications (see, for example, the Java Grande initiative at *http://www.javagrande.org/*). Java's clean and type-safe object-oriented programming model together with its support for parallel and distributed computing make it an attractive environment for writing reliable, large-scale parallel programs. At the same time, many existing Java implementations have inferior performance of both sequential code and communication primitives. Much effort is being invested in improving sequential performance by replacing the original interpreted byte code with just-in-time compilers, native compilers, and specialized hardware [15, 26, 20].

[*]Contact author

For parallel programming on a cluster of workstations, Java's Remote Method Invocation (RMI) provides a convenient paradigm. RMI's design goal is to support highly flexible programming of distributed applications. In current implementations this flexibility comes at the price of a large amount of overhead, a serious impediment for high performance cluster computing.

On modern workstation clusters, existing implementations of RMI are one or more orders of magnitude slower than the communication mechanisms of other parallel languages. On a Pentium Pro/Myrinet cluster, Sun's JDK 1.1.4 implementation of RMI obtains a two-way null-latency of 1228 μsec, compared to 30 μsec for a Remote Procedure Call protocol in C, using a modern user-level communication library. Almost all of the RMI overhead is in software, so the relative overhead increases as networks become faster.

The purpose of this paper is to study why Java RMI is so slow, and to see how close its performance can be brought to that of other parallel languages. The goal of our work is to obtain an order-of-magnitude performance improvement over existing Java RMI implementations. We therefore designed our Java system from scratch, rather than enhance an existing system. Manta uses an optimizing native Java compiler rather than a just-in-time compiler or an interpreter. The Manta run time system is written in C and was designed from scratch to implement method invocation efficiently, in cooperation with the compiler. The compiler generates information that the run time system uses to speed up RMI serialization, thread management, and garbage collection. For example, a major source of serialization overhead is determining the type of objects to be serialized at run time. In Manta, when the type of an object is known, the compiler generates a specialized serializer and inserts a direct call to it. Furthermore, when the compiler can determine that an upcall handler is small and will not block, no new thread is created, but the code is executed on the caller's stack. At the lowest layer, the run time system is implemented on top of an efficient user-level communication layer for Myrinet (rather than on top of TCP), to avoid expensive traps to the operating system.

With these optimizations, the latency obtained by the Manta system on Myrinet is 35 μsec for a null-RMI (an RMI with no arguments and no return value). The best throughput of our system is 20.6 MByte/s. Table 1 shows the two-way null-RMI latencies of Manta (a native Java compiler), Sun's JDK (a byte code interpreter), Sun's JIT (a just-in-time byte code compiler), and Panda (a conventional RPC in C), on three different processors and two different networks. The table shows that Manta is orders of magnitude faster than Sun's RMI, and close to the Panda RPC lower bound.

For most of our optimizations we use insights from earlier work on efficient user-level communication, such as Active Messages [29]. Manta's compiler generated marshaling is similar to Orca's marshaling [2]. The buffering and dispatch scheme is similar to the friendly upcall model [17]. Another optimization is to run small, non-blocking, procedures in the interrupt handler, to avoid expensive thread switches. This technique is first described for Optimistic Active Messages [31]. Instead of kernel-level TCP/IP, Manta uses Panda on top of LFC, a highly efficient user-level communication substrate [4].

The contribution of our work is to show that RMI does not have to be slow. The performance improvement for a null-RMI on Myrinet is a factor of 35, from 1228 to 35 μsec. Manta shows that remote method invocation can be implemented efficiently

System	*Version*	*Processor*	*Network*	*Latency* (μs)
Sun JDK	1.1.3	143 MHz Sparc Ultra 1	Fast Ethernet	3190
Sun JIT	1.1.6	143 MHz Sparc Ultra 1	Fast Ethernet	2240
Sun JDK	1.1.3	300 MHz Sparc Ultra 10	Fast Ethernet	1630
Sun JIT	1.1.6	300 MHz Sparc Ultra 10	Fast Ethernet	1160
Sun JDK	1.1.4	200 MHz Pentium Pro	Fast Ethernet	1905
Manta		200 MHz Pentium Pro	Fast Ethernet	230
Panda	3.0	200 MHz Pentium Pro	Fast Ethernet	222
Sun JDK	1.1.4	200 MHz Pentium Pro	Myrinet	1228
Manta		200 MHz Pentium Pro	Myrinet	35
Panda	3.0	200 MHz Pentium Pro	Myrinet	30

Table 1: Two-way Null-RMI Latency

on high speed networks, making it a suitable technique for high performance parallel computing on a cluster of workstations.

The current version of Manta does not support the full Sun RMI specification. The Java RMI system is designed for distributed computing and has elaborate support for naming and binding to running processes, as well as for security, heterogeneity, and handling of network errors. High performance parallel applications have different requirements than distributed applications. Manta is designed for parallel computing on a homogenous cluster machine, and Manta's RMI does not support Sun's protocol for binding, versioning, security, heterogeneity, and dynamic loading of remote classes. On the other hand, Manta does have the same syntax of a method call and declaration, and has the same call-by-value/call-by-reference semantics of standard and remote objects as Sun's RMI. Support for serialization, multithreading, synchronized methods, and garbage collection is the same (only faster). In addition, we have found it useful to add support for remote creation of new threads and objects, for a different declaration syntax of remote classes, and to relax Sun's stringent rules for the catching of communication exceptions.

The rest of the paper is structured as follows. In Section 2 we first look at related work. An analysis of overhead in RMI is given in Section 3. The implementation is discussed in Section 4. In Section 5 we give a detailed performance analysis of our techniques, using a 128 node Pentium Pro cluster.

2 Related Work

Java supports multiple threads of execution for writing parallel programs on a shared address space machine [18]. In addition, version 1.1 of Java has support for distributed (client/server) programming. Through the RMI mechanism, a program running on one virtual machine can invoke a method of a program running on a different virtual machine [10], similar to a Remote Procedure Call [5]. With RPC the parameter types of a remote procedure typically has to be known at compile time; RMI is more flexible in that it allows subtyping at run time [30] (not yet supported by Manta). The RMI mechanism can also be used for parallel programming. Unfortunately, existing implementations of RMI are too slow to be useful for parallel programming. In this paper we attempt to remedy this.

There are many research projects for parallel programming in Java. *Titanium* [32] 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 Titanium compiler translates Titanium into C. Titanium supports both shared-memory and distributed-memory architectures, and is built on the Split-C/Active Messages back-end.

The JavaParty system [22] is designed to ease parallel cluster programming in Java. In particular, its goal is to run multi-threaded programs with as little change as possible on a workstation cluster. It allows the methods of a class to be invoked remotely by adding a `remote` keyword to the class declaration, removes the need for elaborate exception catching of remote method invocations, and, most importantly, allows objects and threads to be created remotely. Manta's programming model is similar to the JavaParty model. The difference is that our system is designed from scratch for high performance. JavaParty is implemented on top of Java RMI, and thus suffers from the same performance problem as RMI. JavaParty supports object migration, a feature for which we found no need in our experience, and which Manta does not support.

IceT [9] enables users to share Java Virtual Machines across a network. A user can upload a class to another virtual machine using a PVM-like interface. By explicitly calling *send* and *receive* statements, work can be distributed among multiple JVMs. IceT thus provides separate communication primitives, whereas Java RMI (and Manta) use object invocation for communication.

Java/DSM [33] implements a JVM on top of ThreadMarks [14], a distributed shared memory system. No explicit communication is necessary, all communication is handled by the underlying DSM. This has the advantage of transparency, but it does not allow the programmer to make decisions about the parallelism in the program. No performance data for Java/DSM were available to us.

Other related projects are ARMI [23], SPAR [27] and Charlotte [13]. None of these systems, however, are specifically designed to improve the performance of RMI. Our system differs by being designed from scratch to provide high performance, both at the compiler and run time system level. Lessons learned from the implementation of other languages for cluster computing were found to be quite useful. These implementations are built around user level communication primitives, such as Active Messages [29]. Examples are Concert [12], CRL [11], Orca [1, 2], Split-C [8], and Jade [24]. Optimistic Active Messages is described in [31]. The heart of Manta's RTS is based on our experience with fast marshaling and dispatch. Much of this experience has been gathered while porting the Orca system to Myrinet [2].

3 Categories of Overhead in RMI

RMI was designed for distributed programming rather than for high-performance parallel computing. Distributed applications typically run over a slow network (an Ethernet or the Internet) that has a latency of many milliseconds. The popularity of Java has stimulated researchers to investigate whether the RMI model can also be used for parallel programming. On modern hardware, parallel computing is characterized by finer granularity communication, and latencies on the order of microseconds rather than milliseconds.

We will now analyze the performance of Sun's JDK 1.1.4 RMI implementation in order to understand its performance problems. Our cluster of 128 Pentium Pros (running at 200 MHz) is connected by 1.2 Gbit/sec switched Myrinet and 100 Mbit/sec hubbed Fast Ethernet. We have ported the JDK to this system both on kernel-level TCP/IP over Fast Ethernet and on user-level Fast Messages [21] over Myrinet. The source code has been instrumented with timing calls to determine the main sources of overhead of the JDK. Table 2 shows the results for 3 different methods, on Fast Ethernet. The first method takes no parameters; the second one takes an empty object as parameter; the third one takes an object containing an array of 10 Kbytes as parameter. The total overhead of timing calls is less than 10 per cent.

	empty	1 object	2 object	3 object
Serialization	-	686.62	881.41	1120.84
RMI Streams and dispatch	907.255	1140.90	1127.23	1098.71
TCP/IP + kernel	809.242	830.94	814.85	819.64
Total	1719.87	2661.66	2826.84	3042.46

Table 2: Sun JDK 1.1.4, Pentium Pro, FastEthernet; times in μs

The table summarizes the measurements into the three main sources of overhead in the JDK. First and foremost is the serialization method. The RMI serialization code is written in Java, and uses run time object inspection to determine the type of an object. It then traverses the Java Native Interface a number of times to call the actual serializers. The second major source of overhead is due to dispatching, the management of streams and the copying of data between buffers. The JDK is structured around four stream layers (an ObjectStream, a DataStream, a FileStream, and a SocketStream), with their associated copying of data and calling of virtual methods. A third significant source of overhead concerns the low-level primitives: the kernel and network overhead of TCP/IP and the context switches required to service incoming and outgoing messages.

4 Design and Implementation of Manta

The previous section identified three major areas of overhead: serialization, RMI stream management and method dispatch, and low level network. The overhead is caused by byte code interpretation, JNI boundary crossings, copying, software organization, virtual method calls, run time type lookup, and context switching, most of which are eliminated in our scheme by using a compiler. Below we will describe the implementation of our efficient method invocation system in more detail, for each of the three major overhead categories of serialization, RMI streams and dispatch, and low level network. The main idea for serialization is to determine the types of remote objects statically by the compiler, and then to generate specialized serializers. The main idea for RMI dispatch is to use a simpler protocol consisting of fewer stream layers, resulting in less copying and virtual method calls. The main idea for the lowest layer is to use Panda, a highly efficient user-level communication library, instead of TCP/IP, resulting in less copying and context switching.

We will now describe the layers in more detail, starting at the lowest layer. Subsequently, we discuss the Manta compiler and the restrictions of our current implementation.

4.1 Low-level Communication

Most Java RMI implementations are built on top of TCP/IP, which was not designed for parallel processing. Manta is built on top of the Panda communication library [1, 25]. Panda provides efficient communication primitives (message passing, Remote Procedure Call, and broadcast) using an interface that is independent of the underlying machine. The library has been implemented on a variety of platforms. Panda was designed in a modular way, allowing it to exploit low-level primitives of the target machine (e.g., reliable message passing) [25].

On Myrinet, Panda is implemented on top of an efficient network interface protocol called LFC [3, 4], which provides reliable, flow-controlled communication. To avoid the overhead of operating system calls, LFC and Panda run completely in user-space. The Myrinet network device is accessed directly from LFC library primitives. The current implementation of LFC does not provide protection between different user processes: it allows only a single user process to access the network at a given time. (Techniques for protected user-level access do exist, however, and have been implemented in other Myrinet protocols [3].) On Fast Ethernet, Panda is implemented on top of UDP. In this case, the network interface is managed by the kernel. The Panda RPC protocols run in user-space.

The Panda RPC interface is based on an *upcall* model: conceptually a new thread of control is created when a message arrives, which will execute a handler for the message. The interface has been designed to avoid thread switches in simple cases. Unlike active message handlers [29], upcall handlers in Panda are allowed to block (e.g., at a critical section), but a handler is not allowed to wait for another message to arrive. This restriction allows the implementation to handle all messages using a single thread and to avoid context switches for handlers that execute without blocking [16]. We exploit this feature to implement remote invocations of simple Java methods efficiently, as discussed later.

On Myrinet, Panda and LFC also cooperate to reduce the overhead of interrupts for incoming messages. Interrupts (signals) have a high overhead on most operating systems; on our system the overhead is 24 μsec. The optimization we apply is to generate interrupts only if the host processor is busy. When the host is idle (e.g., when it is waiting for an RPC reply) it extracts messages from the network using polling, which has much less overhead than interrupts. LFC allows the host to turn off interrupts from the network interface, which Panda uses to disable network interrupts when the thread scheduler has no runnable threads.

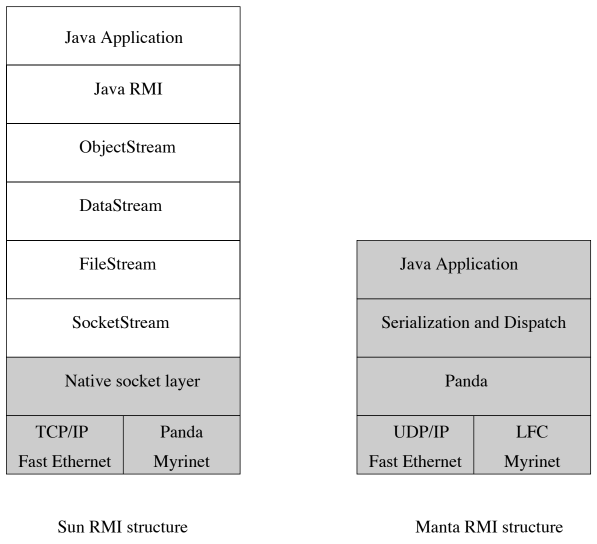

Figure 1: Structure of Sun and Manta RMI

4.2 The RMI Protocol

The Manta RMI protocol is implemented on top of Panda's Remote Procedure Call primitive. The run time system is written entirely in C. It was designed to minimize sources of marshaling and dispatch overhead, such as copying, buffer management, fragmentation, thread switching, and indirect method calls.

Figure 1 gives an overview of the layers in our system and compares it with the layering of the JDK system. The shaded layers denote compiled code, while the white layers are interpreted Java. As can be seen, Manta avoids the stream layers of the JDK. Manta does support programmer's use of Java's stream-classes, but it does not use them for RMI, unless the programmer explicitly writes to them. Instead, RMIs are marshalled directly into a buffer. Moreover, in the JDK these stream layers are written in Java and their overhead thus depends on the quality of the interpreter or JIT. In Manta, all layers are either implemented as compiled C code or compiler generated native code. Also, Manta uses a simple interface from the (compiled) Java application to the RMI implementation (in C), instead of the less efficient JNI interface.

At the sending side, Manta builds a Panda message, which is a contiguous buffer containing an RMI header and the (marshalled) parameters of the invocation. The layout of the buffer is shown in Figure 2. It contains the destination processor, the method to be called (represented as an index into a virtual method table) and a reference to a local stub of the remote object. The *opcode* field is used by the protocol to distinguish different types of messages (e.g., remote invocation, reply from a remote invocation, or creation of a new remote object). Finally, the header contains a flag that is determined by the compiler and that indicates whether a new thread should be created to execute the invocation at the receiving side (see Section 4.4). This technique was first described

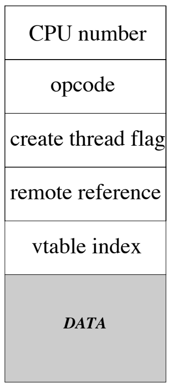

Figure 2: The lay-out of a call message

for OAM [31].

The parameters of the invocation are copied into the buffer (after the header). The protocol marshals the parameters directly into the message buffer, without making any intermediate copies.

At the receiving side, Panda invokes an upcall handler to execute the request. The handler is either invoked from a user-level interrupt (Unix signal handler) or through polling, as described above. The handler first checks whether a new thread of control must be created, by inspecting the *create thread* flag in the RMI header. Next, the handler function or the new thread will unmarshal the parameters, invoke the method, and return a reply to the sending machine.

The RMI protocol cooperates with the garbage collector to keep track of references across machine boundaries. Manta uses a local garbage collector based on a simple mark-and-sweep algorithm. Each machine runs this local collector, using a dedicated thread that is activated by the run time system or the user. The distributed garbage collector is implemented on top of the local collectors, using a reference counting mechanism for remote objects.

Sun's implementation is designed for flexibility, allowing run time loading of new classes. Currently this is not supported in Manta; we trade off some flexibility for efficiency. We are working on compiler support for on demand loading of new classes. Manta already allows dynamic loading of bytecodes, which is implemented by compiling bytecodes dynamically to C and subsequently to binary code. Full interoperability also requires generation of marshaling code, and an interface to Sun's object layout and network protocol.

4.3 The Serialization Protocol

The top layer performs the serialization, or marshaling, of method arguments. It is an important source of overhead of existing RMI implementations. Serialization takes Java objects and converts (serializes) them into an array of bytes. The JDK serialization protocol is written in Java and uses reflection to determine the type of each object during run time. With Manta, all serialization code is generated by the compiler, avoiding the overhead of dynamic inspection.

The protocol performs several optimizations for simple objects. An array whose elements are of a primitive type, for example, is serialized by doing a direct memory-copy into the message buffer, which saves traversing the array. In order to detect duplicate objects, the serialization code uses a hash table containing objects that have already been serialized. If the method does not contain any parameters that are objects, however, the hash table is not created, which again makes simple methods faster. Also, the hash table itself is not transferred over the network; instead, the table is rebuilt on-the-fly by the receiver. Compiler generation of serialization is one of the major improvements of Manta. It has been used before in Orca [1].

4.4 The Compiler

The Manta system uses a native compiler that translates Java directly to executable code, without generating bytecode first. The structure of the Manta compiler is shown in Figure 3. An intermediate code called *Gasm* is used to obtain portability. Gasm is a stack-based low-level intermediate language, extended with registers. The Manta front-end compiles Java into Gasm. Several optimizations are performed on Gasm. Optimizations include: peepholing on intermediate and target language, method inlining by analyzing the class hierarchy, copy and constant propagation, dead-store elimination, jump-to-jump elimination and aggressively replacing stack positions by registers (register promotion). A back-end generates assembly code from Gasm. At present, we have implemented back-ends for the Intel x86 and the Sparc.

The compiler performs several tasks to speed up RMIs. As stated before, all the serialization code is generated by the compiler, to avoid the overhead of run time inspection. Also, the compiler determines for each remote method whether it can be executed without creating a new thread, and whether serialization hash tables are needed. Recall that the Panda RPC interface does not allow upcall handlers to wait for other incoming messages or block on condition synchronization (see Section 4.1). For Java, this restriction implies that the handler should not invoke another remote method, since method invocations are synchronous (blocking). At present, the Manta compiler is conservative and only executes methods in the same thread if the method does not do any method calls at all. Methods are allowed to be synchronized.

To acomplish fast marshaling with correct Java semantics, a shadow vtable is generated by the compiler. For every method in the normal method vtable, a method pointer is maintained here to dispatch to the right unmarshaller for that method. A similar trick is used for objects: every object has two pointers in its vtable to the serializer and deserializer for that object. When a particular object is to be serialized the method pointer is extracted from the objects vtable and called. On deserialization the same procedure is reapplied for every last deserialized object. The serialization and deserialization code (in C) is generated by the compiler and thus has complete information about fields and their types. When a class to be serialized/deserialized is marked “final”, the cost of the virtual function call to the right serializer/deserializer is optimized away, since the correct function pointer can be determined at compile time.

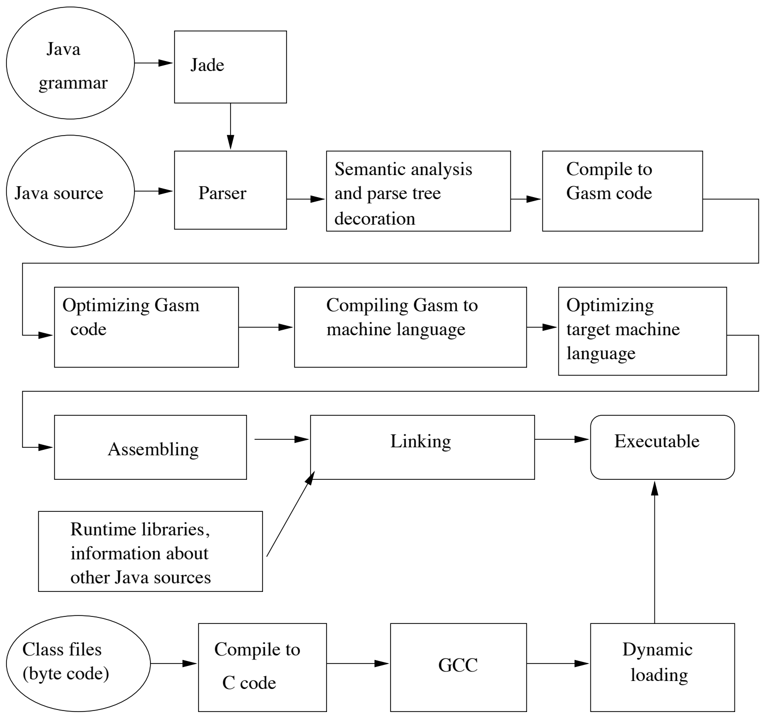
Java grammar
Jade
Java source
Parser
Semantic analysis and parse tree decoration
Compile to Gasm code
Optimizing Gasm code
Compiling Gasm to machine language
Optimizing target machine language
Assembling
Linking
Executable
Runtime libraries, information about other Java sources
Class files (byte code)
Compile to C code
GCC
Dynamic loading

Figure 3: Structure of the Manta Compiler

The Manta compiler is described further in [28].

4.5 Limitations of the Manta RMI Protocol

Our implementation of remote method invocations is designed for high performance on homogeneous cluster computers. Our approach therefore has been to support RMI features that are needed for parallel programming (such as automatic serialization, multithreading, synchronized methods, and garbage collection) and to omit functionality that may decrease performance and is not required for parallel cluster computing. We now discuss in more detail how our RMI system differs from standard Java RMI.

The programming model provided by Manta is similar to that of JavaParty. The main addition to Sun's RMI is that we have direct support for the creation of remote objects and threads (that is, `new` may create an instance on a different JVM). This makes it easier to write parallel programs in a multi-threaded style, where Sun's original RMI is more amenable for client/server-style programs.

In Manta, remote classes can be annotated by a new keyword `remote`, in addition to implementing the predefined interface (`java.rmi.remote`). Exception handling in Manta is less strict than with Sun's RMI. As in JavaParty, the programmer may choose to omit exception handlers for all invocations, in which case communication exceptions become fatal errors. This is easier on the programmer and probably adequate for most programs running on parallel clusters.

The Manta system uses a native compiler, which translates Java directly into executable code, without using bytecodes. Manta has its own object layout and network protocol, making it incompatible with Sun's RMI. Manta does not adhere to Java's "write once, run anywhere" model. It is not yet possible to dynamically load remote methods as described in [30]. Also, in Manta all participating processes must be started at the same time; binding to existing processes is not (yet) supported. Most of the optimizations we apply to improve RMI performance could also be implemented in a JIT. Some of our optimizations, however, require that information is passed from the compiler to the run time system (e.g., serialization routines, information for the garbage collector, and the "create thread" flag). Furthermore, Manta does not support Java's heterogeneity and security model. Adding security managers, however, would add overhead only during the initialization of communication channels, not during their further use.

5 Performance Measurements

In this section, the performance of Manta is compared against the Sun JDK 1.1.4. Experiments are run on a homogenous cluster of 128 Pentium Pro processor. Each node contains a 200 MHz Pentium Pro, 128 MByte of EDO-RAM and a 2.5 GByte IDE disk. All boards are connected by two different networks: 1.2 Gbit/sec Myrinet [6] and Fast Ethernet (100 Mbit/sec Ethernet). The system runs the BSD/OS (Version 3.0) operating system from BSDI. Both Manta and Sun's JDK run over Myrinet and Fast Ethernet. We have created a small user-level layer that implements socket functionality in order to run JDK RMI over Myrinet.

For comparison, we have also run tests on Solaris with the Sun JIT 1.1.6 just-in-time byte code compiler and the Sun JDK 1.1.3 byte code interpreter. These tests were run on two 300 MHz Ultra 10 Sparc's running Solaris 2.5.1 connected by 100 Mbit/s Fast Ethernet. We were unable to run Sun's JIT on BSD/OS.

5.1 Remote Invocation

We first discuss two low-level benchmarks that measure the latency and throughput of RMIs.

5.1.1 Latency

For the first benchmark, we have made a breakdown of the time spent in remote method invocations, using zero to three (empty) objects as parameters, and no return value. The measurements were done by inserting timing calls, using the Pentium Pro performance counters. These counters have a granularity of 5 nanoseconds. The results are shown in Table 3. For comparison, Tables 4, 5 and 6 show the same breakdown for Sun's JDK and JIT on 300 MHz Sparcstations with Fast Ethernet. We have not yet performed a detailed breakdown for Sun's JDK on Myrinet. The empty call times differ slightly from those in table 1, since for these tables more (due to JNI relatively expensive, native) timing calls were inserted. The times are averages over 100,000 RMIs, and are highly consistent.

	empty	1 object	2 objects	3 objects
Serialization	2.6	5.1	7.1	8.8
RMI Overhead	2.6	5.6	9.0	11.9
Panda	30.0	30.0	30.0	30.0
Total	35.2	40.7	46.1	50.7

Table 3: Manta, Pentium Pro, Myrinet; times in μs

	empty	1 object	2 objects	3 objects
Serialization	-	686.62	881.41	1120.84
RMI Overhead	907.255	1140.90	1127.23	1098.71
TCP/IP + kernel	809.242	830.94	814.85	819.64
Total	1719.87	2661.66	2826.84	3042.46

Table 4: Sun JDK 1.1.4, Pentium Pro, FastEthernet; times in μs

The tables show how expensive Sun's RMI serialization and stream and dispatch overhead are, compared to Manta. Even on Fast Ethernet, the software overhead significantly exceeds the network latency.

The simplest case is an empty method without any parameters, the null-RMI. On Myrinet, a null-RMI takes about 35 microseconds. Only 5 microseconds are added to

	empty	1 object	2 objects	3 objects
Serialization	-	564.18	748.78	893.25
RMI Overhead	726.14	758.39	768.04	815.54
TCP/IP + kernel	883.74	917.24	948.27	873.72
Total	1612.67	2242.85	2468.54	2585.83

Table 5: Sun JDK 1.1.3, Sparc Ultra 10, FastEthernet; times in μs

	empty	1 object	2 objects	3 objects
Serialization	-	303.66	403.91	432.17
RMI Overhead	707.84	732.71	766.71	738.07
TCP/IP + kernel	499.91	473.32	496.18	510.97
Total	1210.14	1513.30	1670.48	1685.30

Table 6: Sun JIT 1.1.6, Sparc Ultra 10, FastEthernet; times in μs

the latency of the Panda RPC, which is 30 microseconds. When passing primitive data types as a parameter to a remote call, the latency grows with less than a microsecond per parameter, regardless of the type of the parameter. With an empty call, the Manta serializer performs a small amount of bookkeeping of 2.6 μs. For the Sun JDK and JIT the serialization process is organized differently, and takes less than a microsecond for an empty call.

When one or more objects are passed as parameters in a remote invocation, the latency increases considerably. The reason is that a table must be created by the run time system to detect possible cycles and duplicates in the objects. Separate measurements show that almost all time that is taken by adding an object parameter is spent at the remote side of the call, deserializing the call request (not shown). The marshaling of the request on the calling side, however, is affected less by the object parameters.

We also measured the time to create a new thread for an incoming invocation request, which is required for methods that potentially block (see Section 4.4). Starting a new thread for an invocation costs 16.0 microseconds with the Manta run time system, so a remote call that is executed by a new thread costs at least 51 microseconds. Our optimization for simple methods (that may be synchronized, but may not call `wait()`) thus is useful.

We also analyzed the performance of Manta RMI over Fast Ethernet. A null-RMI for Manta over Fast Ethernet takes 230.2 microseconds, while Java RMI takes 1905 microseconds, so Manta RMI is about 8 times faster. On Myrinet, a null-RMI with the Java JDK takes 1218 microseconds, while Manta takes 35 microsecond, so Manta is a factor 35 faster. The relative Java RMI serialization overhead becomes more significant when the latency of the network decreases.

System	*Version*	*Processor*	*Network*	*Throughput* (MByte/s)
Sun JDK	1.1.3	143 MHz Sparc Ultra 1	Fast Ethernet	0.40
Sun JIT	1.1.6	143 MHz Sparc Ultra 1	Fast Ethernet	1.00
Sun JDK	1.1.3	300 MHz Sparc Ultra 10	Fast Ethernet	1.00
Sun JIT	1.1.6	300 MHz Sparc Ultra 10	Fast Ethernet	4.11
Sun JDK	1.1.4	200 MHz Pentium Pro	Fast Ethernet	0.97
Manta		200 MHz Pentium Pro	Fast Ethernet	7.3
Panda	3.0	200 MHz Pentium Pro	Fast Ethernet	8.1
Sun JDK	1.1.4	200 MHz Pentium Pro	Myrinet	4.66
Manta		200 MHz Pentium Pro	Myrinet	20.6
Panda	3.0	200 MHz Pentium Pro	Myrinet	26.1

Table 7: Throughput

5.1.2 Throughput

The second benchmark we use is a Java program that measures the throughput for a remote method invocation with an array of a primitive type as argument, and no return type. The reply message is empty, so the one-way throughput is measured. In Manta, all arrays of primitive types are serialized with a memcpy(), so the actual type does not matter. The resulting measurements are shown in Table 7.

The table also shows the measured throughput of the Panda RPC protocol, with the same message size as the remote method invocation. Panda achieves a throughput of 37 MByte/s on Myrinet, if no additional copying to use the message is performed. Manta's protocol makes two additional copies of the data: one at the sending side (during serialization) and one at the receiving side (deserialization). Since memory-copies are expensive on a Pentium Pro [7], they decrease the throughput. The table shows Panda throughput when extra copies are performed. As can be seen, Manta RMI achieves a comparable throughput as a Panda RPC that does two extra memcopies. The remaining difference is caused by the overhead of packing and unpacking the call headers and the creation of the Java array object. The measurements also show that for larger array sizes, the memory-to-memory copies have a larger impact on the performance. For array sizes of one megabyte, the throughput is more than halved by the two memory copies.

For the Sun JIT throughput is significantly worse, and even more so for the JDK.

5.2 Compiler versus RTS

An interesting question is whether most of the gain in speed could have been achieved with either the compiler or the run time system.

The large difference in performance between the interpreter and the just-in-time compiler indicates that executing an RMI involves a large amount of Java code. Much of the work that Sun's RMI system performs at run time, Manta does at compile time. The boundary between run time system and compiler has been shifted. This indicates that the answer to the question where most of Manta's gain comes from, is besides the point: A Manta RMI is fast because of the ultra light run time system; but the run time system could be made so thin because most of the functionality is now done by the compiler. The answer has to be that the one needs the other, and, no, the gain in speed could not have been achieved with either the compiler or the run time system.

5.3 Application Performance

In addition to the low-level latency and throughput benchmarks, we have also used three parallel applications to measure the performance of our system. The applications are Successive Overrelaxation, a numerical grid computation, Travelling Salesperson, a combinatorial optimization program, and Iterative Deepening A*, a search program. The applications are described in [19]. We have implemented the programs with TCP/IP sockets, Sun RMI, and JavaParty, and Manta RMI. As can be expected from the low-level benchmarks, Manta's speedup scales better when the number of processors grows. (With constant problem scaling, as the number of processors grows, the proportion of communication time relative to total run time increases, causing a system with a low communication overhead per processor to scale better.) Since Manta has also the best single processor execution times (by a factor of three to seven), Manta consistently has the best parallel execution times by a wide margin.

JavaParty's and Manta's ability to create new remote threads was found to be quite useful, leading to cleaner code and shorter programs than a strict client-server style.

The size of our programs is rather small, a few thousands lines of code in total. So far our application experience confirms the low-level results, although more experience is needed to assess the full impact of faster RMI at application level.

6 Discussion and Conclusion

The low level and application measurements show that Manta's RMI implementation is substantially faster than the Sun JDK and JIT: on Fast Ethernet, the null-latency is improved from 1905 μs to 230 μs, on Myrinet from 1228 μs to 35 μs.

Manta is substantially faster than other implementations at all three layers, serialization, RMI dispatch, and low level communication. Manta uses a new compiler as well as a new run time system. An interesting question is whether most of the gain in speed could have been achieved with either of them—a standard JIT with a better run time system, or a native compiler for Sun's standard RMI sub system. Our measurements showed both elements to be important. Especially the cooperation between compiler and run time system for serialization is of great importance for efficiency.

The focus of our work is on parallel programming on a cluster of workstations, using Java's RMI primitive. RMI is originally designed for distributed (client/server) computing, not for parallel computing. Sun's implementation is designed for flexibility, not for speed.

Other parallel-Java-for-cluster systems are typically ports of existing packages (for example, Java/PVM), or focus on new constructs for better programmability. We focus on speed for an existing Java construct. The low-level benchmarks show that RMI can be as efficient as the best conventional RPC implementations. The preliminary experience with applications has confirmed that the programming model of invoking methods remotely is well suited for parallel programs, especially when the feature of a remote `new` is added.

It turns out that with the right combination of user level messaging and compiler techniques Java's RMI can be made almost as fast as the best C-based RPC implementation. Of these techniques, the three most important are: removing redundant copying in software layers (as our compiler does as much as possible); generating marshaling by the compiler; and using user level communication and Optimistic Active Messages.

Although Manta's RMI is quite usable for writing parallel programs, we would also like to support the distributed programming aspects of Sun's RMI specification. We are currently extending Manta's support for dynamic loading of byte code files, and plan to support binding to existing processes, heterogeneity, and security in the future.

We conclude that remote method invocation can be made to work as fast as other RPC implementations, making Java a viable alternative for high performance parallel programming on a cluster of workstations.

7 Acknowledgements

This work is supported in part by a SION grant from the Dutch research council NWO, and by a USF grant from the Vrije Universiteit. We thank Kees Verstoep for writing a Java socket layer for Myrinet. We thank Raoul Bhoedjang for his unfailing instinct in finding weak spots in our work. His criticism is greatly appreciated. We thank Michael Philippsen for providing us with JavaParty, and for helpful discussions.

References

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

[2] H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, February 1997.

[3] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. Design Issues for User-Level Network Interface Protocols on Myrinet. *IEEE Computer*, 1998.

[4] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Int. Conf. on Parallel Processing*, Minneapolis, MN, August 1998.

[5] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 5(1):39–59, 1984.

[6] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[7] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 214–224, Seattle, WA, USA, June 1997.

[8] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing*, 1993.

[9] P.A. Gray and V.S. Sunderam. IceT: Distributed Computing and Java. *concurrency: practice and experience*, November 1997.

[10] Elliotte Rusty Harold. *Java Network Programming*. O'Reilly, 1997.

[11] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance All-Software Distributed Shared Memory. In *15th ACM Symp. on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, December 1995.

[12] V. Karamcheti and A.A. Chien. Concert - Efficient Runtime Support for Concurrent Object-Oriented. *Supercomputing'93*, pages 15–19, November 1993.

[13] Holger Karl. Bridging the gap between shared memory and message passing. In *ACM Workshop on Java for High-Performance Network Computing*, February 1998.

[14] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 Usenix Conference*, pages 115–131, San Francisco, CA, January 1994.

[15] A. Krall and R. Grafl. CACAO -A 64 bit JavaVM Just-in-Time Compiler. *Concurrency: Practice and Experience*, 1997. http://www.complang.tuwien.ac.at/andi/.

[16] K. Langendoen, R. A. F. Bhoedjang, and H. E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–38, April–June 1997.

[17] K.G. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–37, April–June 1997.

[18] Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 1997.

[19] J. Maassen and R. van Nieuwpoort. Fast Parallel Java. Technical report, 1998.

[20] Giles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa, a mixed offline compiler and interpreter for dynamic class lo ading. Technical report, IRISA / INRIA - University of Rennes, 1997.

[21] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[22] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[23] R.R. Raje, J.I. William, and M. Boyles. An asynchronous remote method invocation (ARMI) mechanism for java. *Concurrency: Practice and Experience*, November 1997.

[24] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[25] T. Rühl, H. Bal, G. Benson, R. Bhoedjang, and K. Langendoen. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 1477–1488, August 9-11 1996.

[26] T. Newsham T.A. Proebsting G. Townsend P. Bridges J.H. Hartman and S.A. Watterson. Toba: Java for applications - a way ahead of time (wat) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[27] K. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: a programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, August 1997.

[28] Ronald Veldema. Jcc, a native Java compiler. Technical report, 1998.

[29] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual Int. Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[30] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, pages 5–7, July–September 1998.

[31] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 217–226, Santa Barbara, CA, July 1995.

[32] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishmurthy, 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*, February 1998.

[33] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.