Java Remote Method Invocation

Pramod Kumar & Ruchi Yadav
Dept. of Information & Technology Dronacharya College of Engineering, Farruhknagar, Gurgaon, India

Abstract
Java offers interesting opportunities for parallel computing. In particular, Java Remote Method Invocation provides an unusually flexible kind of Remote Procedure Call. Unlike RPC, RMI supports polymorphism, which requires the system to be able to download remote classes into a running application. Sun's RMI implementation achieves this kind of flexibility by passing around object type information and processing it at run time, which causes a major run time overhead. Using Sun's JDK 1.1.4 on a Pentium Pro/Myrinet cluster, for example, the latency for a null RMI (without parameters or a return value) is 1228 μse c, which is about a factor of 40 higher than that of a user-level RPC. In this paper, we study an alternative approach for implementing RMI, based on native compilation. This approach allows for better optimization, eliminates the need for processing of type information at run time, and makes a light weight communication protocol possible. We have built a Java system based on a native compiler, which supports both compile time and run time generation of marshallers. We find that almost all of the run time overhead of RMI can be pushed to compile time. With this approach, the latency of a null RMI is reduced to 34 μsec, while still supporting polymorphic RMIs (and allowing interoperability with other JVMs).

Keywords: Invocation, Implementation, Compilation, Latency, Polymorphic.

I. INTRODUCTION

There is a growing interest in using Java for high-performance parallel applications. Java's clean and type-safe object-oriented programming model and its support for concurrency make it an attractive environment for writing reliable, large-scale parallel programs. For shared memory machines, Java offers a familiar multithreading paradigm. For distributed memory machines such as clusters of workstations, Java provides Remote Method Invocation, which is an object-oriented version of Remote Procedure Call (RPC). The RMI model offers many advantages for parallel and distributed programming, including a seamless integration with Java's object model, heterogeneity, and flexibility. Unfortunately, many existing Java implementations have inferior performance of both sequential code and communication primitives, which is a serious disadvantage for high-performance computing.

Much effort is being invested in improving serial code performanceby replacing the original byte code interpretation scheme with just-in-time compilers, native compilers, and specialized hardware. The communication overhead of Java RMI implementations, however, remains a major weakness. RMI is originally designed for client/server programming in distributed (web based) systems, where latencies on the order of several milliseconds are typical. On more tightly coupled parallel machines, such latencies are unacceptable. On our Pentium Pro/Myrinet cluster, for example, Sun's JDK 1.1.4 implementation of RMI obtains a two way null-latency (the latency of an RMI without parameters or a return value) of 1228 microseconds, compared to 30 microseconds for a user level Remote Procedure Call protocol in C. (A null-RMI in Sun's latest JDK, version 1.2 beta, is even slower.) Part of this large overhead is caused by inefficiencies in the JDK. The JDK is built on a hierarchy of stream classes that copy data and call virtual methods. Serialization of method arguments is implemented by recursively inspecting object types until primitive types are reached, and then invoking the primitive serializers a byte at a time. All of this is performed at run time, for each remote invocation. In addition, RMI is implemented on top of IP sockets, which adds kernel overhead (and four context switches on the critical path). Besides inefficiencies in the JDK, a second and more fundamental reason for the slowness of RMI is the difference between the RPC and RMI models. Java's RMI scheme is designed for flexibility and interoperability. Unlike RPC, it allows classes unknown at compile time to be exchanged between a client and a server and to be downloaded into a running program. In Java, an actual parameter object in an RMI can be a subclass of the method's formal parameter type. In polymorphic object-oriented languages, the *dynamic* type of the parameter-object (the subclass) should be used by the method, not the static type of the formal parameter.

II. DESIGN AND IMPLEMENTATION OF MANTA

This section will discuss the design of the Manta system (including the unimplemented parts) and describe the current implementation status of the system. We will focus on the Manta RMI implementation.

1. Manta Structure

Since Manta is designed for high-performance parallel computing, it uses a native compiler rather than a JIT. The most important advantage of a native compiler is that it can do more aggressive optimizations and therefore generate better code. To support interoperability with other JVMs, however, Manta also has to be able to process the byte code for the application, and contains a run-time byte-code-to-native compiler. The Manta system is illustrated in Figure 1. The box in the middle describes the structure of a node running a Manta application. Such a node contains the executable code for the application and (de)serialization routines, both of which are generated by Manta's native compiler. A Manta node can communicate with another Manta node (the box on the left) through a fast RMI protocol (using Manta's own serialization format); it can communicate with another JVM (the box on the right) through a JDK-compliant protocol (using Sun's serialization format). Determining which protocol to use is done with an initial probe RMI, which is only recognized by a Manta application, not by a JVM. A Manta-to-Manta RMI is performed with Manta's own fast protocol, which is described in detail in the next subsection. This is the common case for high performance parallel programming, for which Manta is optimized. Manta's serialization and deserialization protocols support heterogeneity. A Manta-to-JVM RMI is performed with a slower protocol that
is compatible with Sun's RMI standard. Manta uses generic routines to (de)serialize the objects to or from Sun's format. These routines use runtime type inspection

(reflection), and are similar to Sun's protocol. The routines are written in C (as is all of Manta'sun time system) and execute more efficiently than Sun's protocol, which is written mostly in Java. A Manta application must be able to work with byte codes from other nodes, to implement polymorphic RMIs with JVMs. When a Manta application requests a byte code from a remote process, Manta will invoke its byte code compiler to generate the metaclasses, the serialization routines, and the object code for the methods (as if they were generated by the Manta source code compiler).

2. Serialisation and Communication

RMI systems can be split into three major components: low-level communication, the RMI protocol (stream management and method dispatch), and serialization. Below, we discuss how Manta implements this functionality.

Low – level communication:-

Java RMI implementations are built on top of TCP/IP, which was not designed for parallel processing. Manta uses the Panda communication library, which has efficient implementations on a variety of networks. On Myrinet, Panda uses the LFC communication system . To avoid the overhead of operating system calls, LFC and Panda run in user space. On Fast Ethernet, Panda is implemented on top of UDP. In this case, the network interface is managed by the kernel, but the Panda RPC protocols run in user space.

The RMI protocol:-

The run time system for the Manta RMI protocol is written in C. It was designed to minimize serialization and dispatch overhead, such as copying, buffer management, fragmentation, thread switching, and indirect method calls. Figure 2 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 (or JIT-compiled) Java. Manta avoids the stream layers of the JDK. Instead, RMIs are serialized directly into a Panda 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.

The Serialisation protocol:-

The serialization of method arguments 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. Serialization code for most RMI calls is generated at compile time. Only serialization code for polymorphic RMI calls that are not locally available is generated, by the Manta compiler, at run time. The overhead of this run time code generation is incurred only once, the first time the new class is used as a polymorphic argument to some method invocation.

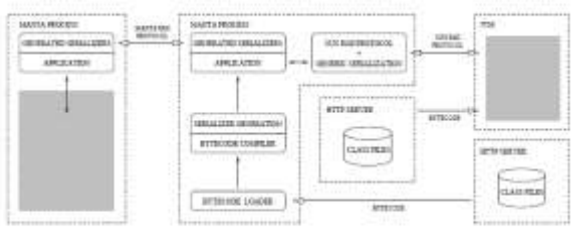

Figure 1: Manta/JVM Interoperability

III. PERFORMANCE MEASUREMENT

In this section, the performance of Manta is compared against the Sun JDK 1.1.4. Experiments are run on a homogeneous cluster of Pentium Pro processors. Each node contains a 200 MHz Pentium Pro and 128 MByte of EDO-RAM. 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 and RedHat Linux version 2.0.36. Timing differences between BSD and Linux are small to negligible. Except where otherwise noted, the numbers reported are from runs on BSD. 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, on top of Illinois Fast Messages (FM). FM's round-trip latency is 4 μs higher than that of LFC.

Latency:-

The simplest case is an empty method without any parameters, the null-RMI. On Myrinet, a null-RMI takes about 34 μs. Only 4 microseconds are added to the latency of the Panda RPC, which is 30 μs. 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 (this is not shown in the table). When one or more objects are passed as parameters in a remote invocation, the latency increases with several microseconds. 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 serialization of the request on the calling side, however, is affected less by the object parameters.

Throughput:-

The table also shows the measured throughput of the Panda RPC protocol, with the same message size as the remote method invocation. Two versions of Panda are shown. The basic version, with which almost all measurements in this paper are performed, is Panda 3.0. On Myrinet we have also performed measurements with Panda 4.0, which supports a scatter/gather interface. This scatter/ gather interface makes it possible to remove some copying of user data from the critical path, resulting in an improved throughput. Unfortunately, dereferencing the scatter/gather vector involves extra processing, which increases the latency somewhat. Panda 3.0 achieves a throughput of 55.7 MByte/s on Myrinet, which is much higher than the throughput for Manta (20.6 MByte/s).

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 Over relaxation (a numerical grid computation), Traveling Salesperson Problem (a combinatorial optimization program), and Iterative Deepening A* (a search program). For TSP we used a 15 city problem, for SOR a 2048 *512 matrix, for IDA* we solved a random instance of a sliding tile puzzle (with solution length 56). The applications are described in more detail in [20]. We have implemented the programs with Sun RMI 1.1.4 (on Fast Ethernet) and Manta/Panda 3.0 RMI (on Fast Ethernet and Myrinet). Figure 6 shows run times, in seconds, for the serial program, and for runs of the parallel program, on 1 and 16 processors. Note the different scale for the 16 processor run. The programs are run on the Pentium Pros on BSD.

IV. RELATED WORK

Many projects for parallel programming in Java exist (see, for example, the JavaGrande web page at http://www.javagrande.org/). Titanium 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. It is built on the Split-C/Active Messages back-end. The JavaParty system 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. JavaParty is implemented on top of Java RMI, and thus suffers from the same performance problem as RMI. Java/DSM implements a JVM on top of TreadMarks, adistributed shared memory system. No explicit communication is necessary, all communication is handled by the underlying DSM. No performance data for Java/DSM were available to us. Breget al study RMI performance and interoperability. Krishna swamy et al improve RMI performance somewhat by using caching and UDP instead of TCP. Sampemane et al describe how RMI can be run over Myrinet using the socket Factory facility. Gokhale et al discuss high-performance computing issues for CORBA. Hirano et al provide performance figures of RMI and RMI-like systems on Fast Ethernet. Our system differs by being designed from scratch to provide high performance, both at the compiler and run time system level. For the non-polymorphic RMI part, Manta's compiler-generated serialization is similar to Orca's serialization. The buffering and dispatch scheme is similar to the single-threaded upcall model. Small, non-blocking, procedures are run in the interrupt handler, to avoid expensive thread switches.

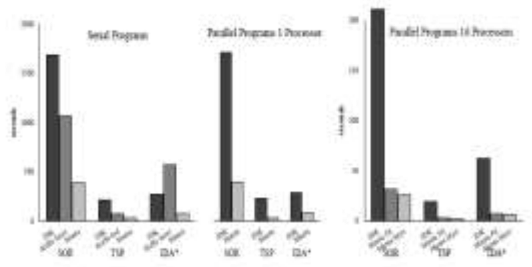

Figure 6: Application Run Time

V. CONCLUSION

We have built a new compiler-based Java system (Manta) that was designed from scratch to support efficient Remote Method Invocations on parallel computer systems. Performance measurements show that Manta's RMI implementation is substantially faster than the Sun JDK and JIT. For example, on Fast Ethernet, the null latency is improved from 1711 μs (for the JDK) to 233 μs, on Myrinet from 1228 μs to 34 μs, in both cases only a few microseconds slower than a C-based RPC. The gain in efficiency is due to three factors: the use of compile time type information to generate specialized serializers; a more streamlined and efficient RMI protocol; and the usage of faster communication protocols. RMI is originally designed for flexible distributed (client/server) computing, and allows subclasses to be downloaded into a running program. Sun's implementation handles serialization, dispatch and buffer management at run time. It is designed for flexibility, not speed. Our system uses compile time information to make the run time protocol as lean as possible, so that processing it will be fast. Flexibility is achieved by recompiling classes and generating serializers as and when they are needed. Our implementation is designed for speed, yet preserves the polymorphism of RMI. We find that with the right combination of user level messaging, compile time type information, and run time compilation, Java's RMI can be made almost as fast as a C-based RPC implementation while retaining the flexibility of RMI, making Java a viable alternative for high performance parallel programming.

VI. ACKNOWLEDGEMENT

This work is supported in part by a USF grant from the Vrije Universiteit. Aske Plaat is supported by a SION grant from the Dutch research council NWO. We thank Kees Verstoep for writing a Java socket layer for Myrinet on FM. Ceriel Jacobs and Rutger Hofman implemented and debugged a substantial part of the Manta system. We thank Raoul Bhoedjang for his keen criticism on this work. We thank Michael Philippsen for providing us with JavaParty, and for helpful discussions. We thank Thilo Kielmann for discussions on polymorphism in distributed object oriented languages and for his feedback on an earlier draft of this paper. We thank the anonymous referees for helpful feedback.

VII. REFERENCES

[1] H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, T. R¨uhl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. Of 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] B. Bershad, S. Savage, P. Pardyak, E. Gun Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility,Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

[4] R. A. F. Bhoedjang, T. R¨uhl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60 , November 1998.

[5] R.A.F. Bhoedjang, T. R¨uhl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Int. Conf. on Parallel Processing*, pages 381–390, Minneapolis, MN, August 1998.