Extending Java's Communication Mechanisms for Multicore Processors

George C. Wells
Department of Computer Science
Rhodes University
Grahamstown, South Africa
G.Wells@ru.ac.za

ABSTRACT

With the current trend towards the increased use of multicore processors, there is a growing need for simple, efficient parallel programming mechanisms. While Java has good support for multithreaded and distributed application development, our research into tuple-space systems for multicore processors highlighted a gap in the concurrency facilities available in Java. This arises in the context of independent applications (running in separate virtual machines) that need to synchronise their activities or communicate with each other. There are several possible solutions to this problem, ranging from extensions to the language and/or runtime environment through to the use of distributed programming methods. Using the latter introduces considerable performance overheads, and so we explored the use of the Java Native Interface in order to take advantage of the interprocess communication (IPC) facilities provided by the underlying operating system. The analysis and comparison of the performance of the standard approaches and our prototype library suggest that there are real benefits to be gained by alternative approaches to the provision of IPC mechanisms for independent Java programs executing on multicore systems. We hope that these findings will spur further investigation of this problem and other possible solutions.

1 Introduction

In recent years there has been a dramatic shift in the trends of computer architecture development, as processors have adopted symmetric multiprocessor (SMP) techniques, exemplified by the increasing prevalence of multicore processors (with increasing numbers of cores). This shift has serious implications for software development practises, forcing programmers to adopt parallel programming techniques. However, parallel programming is not simple — as Jim Larus notes "The popular [parallel] programming models ... are performance-focused, error-prone abstractions that developers find difficult to use"[1].

Java provides a wide range of parallel and distributed programming techniques. There has always been good support for *multithreaded* programs. At the other end

of the parallel–distributed programming spectrum, Java has strong support for various distributed programming models. However, there is a distinct gap in the concurrency tools available in Java when it comes to *interprocess communication* between Java programs running on a shared-memory system in separate virtual machines. This system configuration can be extremely useful in some situations, but is currently only supported by means of distributed programming mechanisms, using the "loop-back" network connection. Our recent research has involved systems for multicore processors, using separate Java processes. When we encountered disappointing performance we were prompted to investigate this issue, and have subsequently developed initial prototypes of a possible solution. These problems, possible solutions and our prototypes are discussed in this paper.

2 Concurrency in Java

Java has always provided comprehensive support for multithreaded applications, with coordination provided through object-level locking[2]. This model uses a shared-address-space model — specifically, the communicating threads must be executing in the context of a single Java Virtual Machine (JVM). More recently, the basic multithreaded facilities offered by Java were extended through the provision of the Java Concurrency Utilities[3]. With these sophisticated facilities available, there is very good support for the development of sophisticated parallel/multithreaded applications in a shared-memory, shared-address-space environment.

Java also provides very good support for distributed applications running across networks. At the lowest level, Java provides basic network classes that allow for the creation and use of sockets. In conjunction with I/O streams and object serialisation, these provide a simple, but flexible and powerful communication mechanism. At a slightly higher level of abstraction, the Enterprise Edition provides a comprehensive messaging API, known as the Java Message Service (JMS).

One of the most well-known forms of distributed computing is remote procedure/method calling. In Java this is provided by both Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA).

At one of the highest levels of abstraction, Java provides support for object- or tuple-space systems through JavaSpaces[4], which is based on the Linda coordination language developed at Yale[5]. As a very high-level abstraction of the coordination activities (i.e. communication and synchronisation) of a distributed application, Linda is very easy to use, but performance may be problematic. The development of Linda systems in Java and their optimisation has long been a focus of ours[6], and it was our research in this area that led to the investigation described in this paper.

3 The Problem

The trend towards multicore processors led us to develop an implementation of our Linda system (eLinda) for such systems. A vital part of any Linda system is the component that manages the data. In our current system this is implemented as a "server", run as a stand-alone process. There are many good reasons for this, not least that it provides a very useful separation between the eLinda system and the client applications making use of it. Besides the logical separation of concerns that this architecture provides, it also helps address security or reliability concerns, as the interaction between a client application and the eLinda system is limited to the explicit communication between the processes. For example, the eLinda system cannot

access any data that is not explicitly passed to it, and exceptions and errors that might arise in the eLinda system do not directly impact the client application(s).

We believe that as the use of multicore processors increases it will be increasingly important to support development of complex applications composed of separate processes. This application architecture bridges the gap between multithreaded applications, and distributed applications. However, at present there are few options available for communication between separate processes in Java. The only widely-available solution is to make use of Java's distributed programming facilities (using the loop-back network). When we used this for the eLinda system, the limitations of this approach became apparent, especially in respect of performance. Our intuition suggested that messages were having to work through the levels of the TCP/IP communication stack, much of which is irrelevant to communication between processes executing on a single computer system.

3.1 Possible Solutions

There are a few possible solutions to this problem, all of which ultimately rely on using the mechanisms provided by the underlying operating system. A simple approach to exploiting the underlying operating system's IPC facilities is through the use of the Java Native Interface (JNI)[7]. This approach was adopted for our initial investigation, which forms the basis of this paper. The benefits and drawbacks of this approach are discussed in more detail later in this paper, but an obvious (and significant) disadvantage is the loss of application portability.

An alternative approach would be to modify the Java Virtual Machine and the Java compiler to provide direct support for IPC. This would be considerably more complicated, and would require modifications to the Java language, and to the JVM. However, it would probably bring additional performance benefits, and could help address the portability problems inherent in using JNI.

4 The IPC Prototypes

Unix-based operating systems provide a wide range of IPC mechanisms. Specifically, the release of Unix System V in the 1980s introduced the so-called System V IPC facilities. These include sophisticated message queues, semaphore sets and shared memory segments. In addition to these, Unix systems support the concept of *pipes*, and more specifically *named pipes*, for IPC. An initial prototype library was developed (called *LinuxIPC*), providing access to these IPC facilities using JNI, for Ubuntu 8.04. This was later ported to the Solaris 10 operating system (called *SolarisIPC*). Both packages provide relatively complete access to the underlying operating system's IPC mechanisms, together with other useful support functions.

In order to simplify the use of these facilities, further classes were developed that implemented *I/O streams* using message queues, and shared memory (synchronised using semaphores). These greatly simplify the use of the IPC packages as they may be used in conjunction with Java's data-formatting and object serialisation streams. The IPC stream classes also provide a useful degree of abstraction, isolating applications from almost all of the details of the IPC system calls. The initial shared memory stream implementation was done using the semaphores and shared memory directly, but the performance was found to be poor, due to the overheads of making native calls. A customised native class was developed that integrated the use of the shared memory and the semaphores into a single native method, thus minimising the number of calls to native methods. This provided a useful performance improvement,

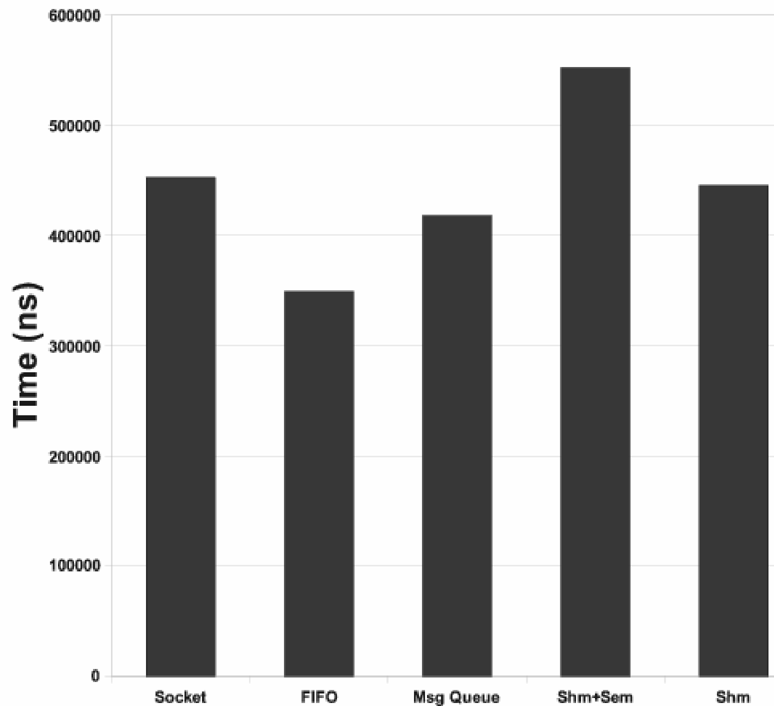

Figure 1: Simple Benchmark Results for Solaris.

evident in the results below.

5 Results

Results for a dual-core Intel processor and Ubuntu were presented previously[8]. These have now been extended with results for an eight-core UltraSPARC T2 processor running Solaris (Sun T6320 server with Solaris 10). These results illustrate the problems inherent with the use of network-based IPC.

The results shown in Figure 1 are for a simple communication benchmark, and highlight the relative efficiency of each of the different IPC methods (network sockets, named pipes/FIFOs, message queues, shared memory and semaphores, and optimised shared memory streams), comparing them with network sockets. The times reported are for a round-trip message between two processes, carrying minimal data. As is clear from the figure, explicitly using sockets and semaphores is less efficient, but the other forms of IPC are more efficient than using sockets. The use of named pipes is by far the most efficient of the mechanisms, which is not surprising, as it only uses JNI in order to create the named pipe, whereafter it is accessed using standard file streams. As can also be seen from these results, the integrated shared memory streams are more efficient than the version using explicit calls to the shared memory and semaphore facilities, as discussed in Section 4 (the improvement is 19.4%).

Performance was also investigated for varying volumes of data (Figure 2). These results follow the expected pattern of the time taken to send a message increasing with the data volume. Notably, the named pipe version is also the most efficient for all message sizes, by a significant margin. However, the network-based version is more efficient than the other forms of IPC for larger messages.

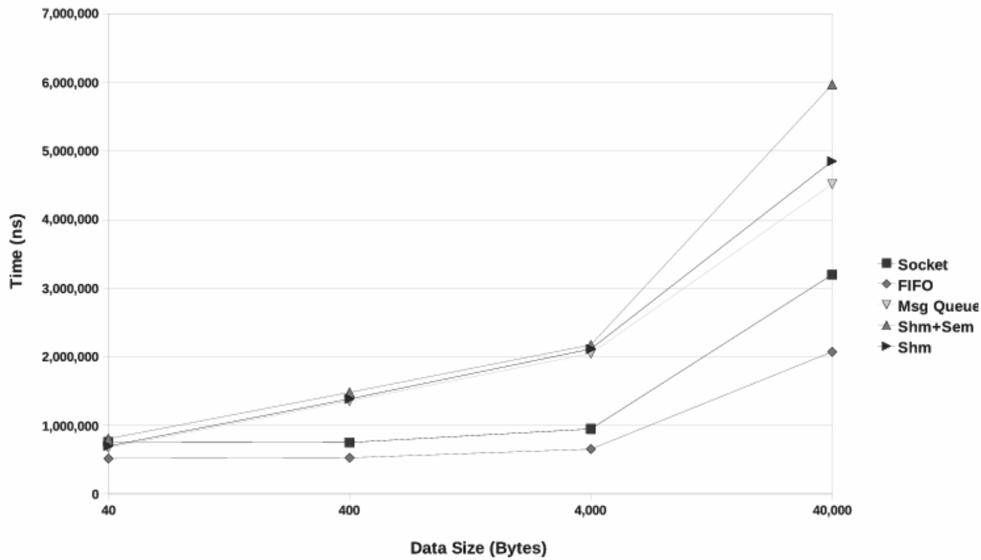

Figure 2: Results for Varying Size Data Transfers.

6 Discussion and Conclusions

The results presented above clearly indicate that network sockets are not the most efficient mechanism for providing IPC in Java, thus confirming our initial concerns about this approach. The results also indicate that named pipes are the most efficient form of communication by a significant margin. This is particularly pleasing because using named pipes makes minimal use of JNI, thus minimising the impact on program portability[1].

6.1 Possible Extensions and Future Research

The research described in this paper presents a preliminary overview of the problems involved in IPC for Java processes, and a prototype of one possible solution. Notably, our work until now has been restricted to Unix-based operating systems. An investigation of the IPC facilities available under Windows would be useful, leading to the development of a JNI-based solution. In particular, this would allow for an initial assessment of the performance of native IPC facilities compared with socket-based communication under Windows. Similar research for Mac OS X and other common operating systems would also be useful in terms of characterising the extent to which there is a need for alternatives to socket-based communication for Java processes. Such a survey of the IPC facilities offered by different operating systems would also be useful in terms of establishing what, if any, mechanisms are common to all widely-used operating systems. If a common subset of IPC facilities could be identified, a Java package could be developed for a common abstraction, providing portability at the source code level.

As mentioned in Section 3.1, a potentially better solution than using JNI would be to extend the Java language with IPC operations. This would provide a great deal of power and flexibility, and could take almost any form desired. In particular, some form of light-weight remote-object-access protocol (similar to RMI, but intended only

[1] The use of JNI may even be avoided completely, as the named pipes can be created independently.

for IPC in an SMP environment) would be very useful, as it would allow programmers to use familiar, object-oriented techniques to build parallel applications with good support for separation of concerns, and security.

Whatever the form of the final solution, we believe that the current trend towards the wide-spread use of SMP architectures will continue, and will provide significant challenges for the development of software able to exploit the potential performance of these systems. While Java's multithreading facilities provide an excellent solution to many problems, they do not provide adequate isolation of semi-independent program components. Furthermore, the use of network-based mechanisms has been shown to offer poor performance for IPC in an SMP environment. The research presented here provides an initial characterisation of these issues, and some indication of the potential for improved performance. We hope the Java community will start to explore these issues more widely, and that ultimately a portable, generic and high-performance solution will be found.

Acknowledgments

This research was performed while visiting the Department of Computer Science at the University of California Davis at the kind invitation of Dr. Raju Pandey. Access to the Sun multicore processor hardware for the Solaris testing was generously provided by Sun Microsystems through the Sun Partner Advantage Program. This project was funded by the South African National Research Foundation (NRF). Financial support was also received from the Distributed Multimedia Centre of Excellence (funded by Telkom, Comverse, Tellabs, Stortech, Amatole Telecom Services, Bright Ideas 39 and THRIP), and from Rhodes University.

References

[1] James Larus. Spending Moore's dividend. *Commun. ACM*, 52(5):62–69, 2009.

[2] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Prentice Hall, 2nd edition, 1999.

[3] Java Community Process. JSR 166: Concurrency utilities. September 2004.

[4] Philip Bishop and Nigel Warren. *JavaSpaces in Practice*. Addison Wesley, 2002.

[5] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[6] G.C. Wells, A.G. Chalmers, and P.G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16:1005–1022, August 2004.

[7] Sun Microsystems, Inc. Java native interface 5.0 specification. 2003.

[8] G.C. Wells. Interprocess communication in Java. In H.R. Arabnia, editor, *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09)*, pages 407–413, Las Vegas, July 2009. CSREA Press.