A New Distributed JVM for Cluster Computing

Marcelo Lobosco[1], Anderson Silva[1], Orlando Loques[2], and Claudio L. de Amorim[1]

[1] Laboratório de Computação Paralela, PESC, COPPE, UFRJ
Bloco I-2000, Centro de Tecnologia, Cidade Universitária, Rio de Janeiro, Brazil
`{ lobosco, faustino, amorim }@cos.ufrj.br`
[2] Instituto de Computação, Universidade Federal Fluminense
Rua Passo da Pátria, 156, Bloco E, 3º Andar, Boa Viagem, Niterói, Brazil
`loques@ic.uff.br`

Abstract. In this work, we introduce CoJVM, a new distributed Java run-time system that enables concurrent Java programs to efficiently execute on clusters of personal computers or workstations. CoJVM implements Java's shared memo-ry model by enabling multiple standard JVMs to work cooperatively and transpa-rently to support a single distributed shared-memory across the cluster's nodes. CoJVM requires no change to applications written in standard Java. Our experi-mental results using several Java benchmarks show that CoJVM performance is considerable with speed-ups ranging from 6.1 to 7.8 for an 8-node cluster.

1 Introduction

One of the most interesting features of Java [1] is its embedded support for concurrent programming. Java provides a native parallel programming model that includes support for multithreading and defines a common memory area, called the heap, which is shared among all threads that the program creates. To treat race conditions during con-current accesses to the shared memory, Java offers to the programmer a set of synchronization primitives, which are based on an adaptation of the classic monitor model. The development of parallel applications using Java's concurrency model is restricted to shared-memory computers, which are often expensive and do not scale easily. A com-promise solution is the use of clusters of personal computers or workstations. In this case, the programmer has to ignore Java's support for concurrent programming and instead use a message-passing protocol to establish communication between threads. However, changing to message-passing programming is often less convenient and even more complex to code development and maintenance. To address this problem, new distributed Java environments have been proposed. In common, the basic idea is to extend the Java heap among the nodes of the cluster, using a distributed shared-memory approach. So far, only few proposals have been implemented, and even less are compliant with the Java Virtual Language Specification [2]. Yet, very few reported good performance results [9] and presented detailed performance analysis [17].

In this paper, we introduce the design and present performance results of a new Java environment for high-performance computing, which we called the **CoJVM**

(**Co**operative **J**ava **V**irtual **M**achine) [3]. CoJVM's main objective is to speed up Java applications executing on homogeneous computer clusters. CoJVM relies on two key features to improve application performance: 1) the HLRC software Distributed Shared Memory (DSM) protocol [4] and 2) a new instrumentation mechanism [3] to the Java Virtual Machine (JVM) that enables new latency-tolerance techniques to exploit the application run-time behavior. Most importantly, the syntax and the semantics of the Java language are preserved, allowing programmers to write applications in the same way they write concurrent programs for the single standard JVM. In this work, we evaluate CoJVM performance for five parallel applications: Matrix Multiplication (MM), SOR, LU, FFT, and Radix. The connected figures show that all benchmarks we tested achieved good speedups, which demonstrate CoJVM effectiveness. Our main contributions are: a) to show that CoJVM is an effective alternative to improve performance of parallel Java applications for cluster computing, b) to demonstrate that scalable performance of Java applications for clusters can be achieved without any syntax or semantic change in the language; and c) to present detailed performance analysis of CoJVM for five parallel benchmarks.

The remainder of this paper is organized as follows. Section 2 presents the HLRC software DSM system. Section 3 describes Java support for multithreading and synchronization, and the Java memory model. In section 4, we review some key design concepts of CoJVM. In section 5, we analyze performance results of five Java parallel applications executed under CoJVM. In section 6, we describe some related works. Finally, in section 7, we draw our conclusions and outline ongoing works.

2 Software DSM

Software DSM systems provide the shared memory abstraction on a cluster of physically distributed computers. This illusion is often achieved through the use of the virtu-al memory protection mechanism [5]. However, using the virtual memory mechanism has two main shortcomings: (a) occurrence of false sharing and fragmentation due to the use of the large virtual page as the unit of coherence, which leads to unnecessary communication traffic; and (b) high OS costs of treating page faults and crossing pro-tection boundaries. Several relaxed memory models, such as LRC [6], have been pro-posed to alleviate false sharing. In LRC, shared pages are write-protected so that when a processor attempts to write to a shared page an interrupt will occur and a clean copy of the page, called the twin, is built and the page is released to write. In this way, modi-fications to the page, called *diffs*, can be obtained at any time by comparing current co-py with its twin. LRC imposes to the programmer the use of two explicit synchroniza-tion primitives: acquire and release. In LRC, coherence messages are delayed until an acquire is performed by a processor. When an acquire operation is executed the acqui-rer receives from the last acquirer all the write-notices, which correspond to modifica-tions made to the pages that the acquirer has not seen according to the happen-before-1 partial order [6]. HLRC introduced the concept of home node, in which each node is responsible for maintaining an up-to-date copy of its owned pages; then, the acquirer can request copies of modified pages from their home nodes. At release points, *diffs* are computed and sent to the page's home node, which reduces memory requirements in home-based DSM protocols and contributes to the scalability of the HLRC protocol.

3 Java

In Java, threads programming is simplified since it provides a parallel programming model that includes support for multithreading. The package java.lang offers the Thread class that supports methods to initiate, to execute, to stop, and to verify the state of a running thread. In addition, Java also includes a set of synchronization primitives and the standard semantics of Java allow the methods of a class to execute concurrently. The synchronized reserved word, when associated with methods, specifies that they can only execute in a mutual-exclusion mode. The JVM specifies the interaction model between threads and the main memory, by defining an abstract memory system, a set of memory operations, and a set of rules for these operations [2]. The main memory stores all program variables and is shared by the JVM threads. Each thread operates strictly on its local memory, so that variables have to be copied first from main memory to the thread's local memory before any computation can be carried out. Similarly, local results become accessible to other threads only after they are copied back to main memory. Variables are referred to as master or working copy depending on whether they are located in main or local memory, respectively. The copying between main and local memory, and vice-versa, adds a specific overhead to thread operation. The replication of variables in local memories introduces a potential memory coherence hazard since different threads can observe different values for the same variable. The JVM offers two synchronization primitives, called monitorenter and monitorexit, to enforce memory consistency. In brief, the model requires that upon a monitorexit operation, the running thread updates the master copies with corresponding working copy values that the thread has modified. After executing a monitorenter operation, a thread should either initialize its work copies or assign the master values to them. The only exceptions are variables declared as volatile, to which JVM imposes the sequential consistency model. The memory management model is transparent to the programmer and is implemented by the compiler, which automatically generates the code that transfers data values between main memory and thread local memory.

4 The Cooperative Java Virtual Machine

CoJVM [3] is a DSM implementation of the standard JVM and was designed to efficiently execute parallel Java programs in clusters. In CoJVM, the declaration and synchronization of objects follow the Java model, in which the main memory is shared among all threads running in the JVM [2]. Therefore in CoJVM all declared objects are implicitly and automatically allocated into the Java heap, which is implemented in the DSM space. Our DSM implementation uses the HLRC protocol for two main reasons. First, it tends to consume less memory and its scalable performance is competitive with that of homeless LRC implementations [4]. Second, the HLRC implementation [7] al-ready supports the VIA [8], a high-performance user-level communication protocol. Surdeanu and Moldovan [9] have shown that the LRC model is compliant with the Java Memory Model for data-race-free programs. Although HLRC adopts the page as the unit of granularity, we are not bound to that specific unit.

CoJVM benefits from the fact that Java already provides synchronization primitives: synchronized, wait, notify and notifyAll. The programmer with the use of these primitives can easily define a barrier or other synchronization constructs, or invoke a native routine. CoJVM supports only Java standard features, and adds no extra synchronization primitive. Since the declaration and synchronization of objects in the DSM follow the Java model and no extra synchronization primitive is added in our environment, a standard concurrent application can run without any code change. Threads created by the application are automatically moved to a remote host, and data sharing among them are treated following the language specification in a transparent way. Currently, CoJVM allows one single thread per node due to HLRC's restriction but we plan to solve this problem soon.

5 Performance Evaluation

Our hardware platform consists of a cluster of eight 650 MHZ Pentium III PCs running Linux 2.2.14-5.0. Each processor has a 256 KB L2 cache and each node has 512 MB of main memory. Each node has a Giganet cLAN NIC connected to a Giganet cLAN 5300 switch. This switch uses a thin tree topology and has an aggregate throughput of 1.25 Gbps. The point-to-point bandwidth to send 32 KB is 101 MB/s and the latency to send 1 byte is 7.9µs.

Table 1. Sequential times of applications, in seconds

Program	Program Size	Seq. Time - JVM	Seq. Time - CoJVM	Overhead
MM	1000x1000	485.77	490.48	0.97 %
LU	2048x2048	2,345.88	2,308.62	0 %
Radix	8388608 Keys	24.27	24.26	0 %
FFT	4194304 Complex Doubles	227.21	228.21	0.44 %
SOR	2500x2500	304.75	306.46	0.56 %

Table 2. Performance of sequential applications: JVM (first figure) X CoJVM (second figure)

Counter	SOR	LU	Radix	FFT
% Load/Store Inst	75.3 / 73.8	86.8 / 90.1	87.4 / 86.0	91.2 / 91.2
% Miss Rate	1.9 / 2.3	1.2 / 6.1	1.2 / 1.2	1.7 / 1.7
CPI	1.83 / 1.83	2.78 / 2.73	1.76 / 1.76	1.85 / 1.85

To quantify the overhead that CoJVM imposes due to modifications of the standard JVM, we ported four benchmarks from the SPLASH-2 benchmark suite [10]: SOR, LU, FFT and Radix, and developed one, MM. The sequential execution times for each application are shown in Table 1. Each application executed 5 times, and the average execution time is presented. The standard deviation for all applications is less than 0,01%. For all applications, the overheads are less than 1%. We also instrumented both machines with the performance counter library PCL [11], which collects at processor level, run-time events of applications executing on commercial microprocessors. Table 2 presents the CPI (cycles per instruction) of sequential executions of SOR, LU, Radix, and FFT on the Sun JDK 1.2.2 and on CoJVM (which is based on Sun JDK 1.2.2 implementation). The table also presents the percentage of load/store instructions and data cache level 1 miss rates. Table 2 shows that the percentage of load/store instructions does not change for FFT, but increases for LU and decreases slightly for SOR and Radix. However, the increase in the number of memory accesses in LU does not have a negative impact on miss rate. Indeed, the miss rate for CoJVM is half of the miss for the standard JVM. CoJVM modifications to the standard JVM resulted in lower CPI rate for LU while for the other applications, both CPI were equal.

Table 3 shows application speedups. We can observe that MM achieved an almost linear speedup while the other applications attained good speedups. Next, we analyze application performance in detail. In particular, we will compare CoJVM scalable performance against that of HLRC running the C version of the same benchmarks.

Table 3. Speedups

Program	2 Nodes	4 Nodes	8 Nodes
MM	2.0	4.0	7.8
LU	2.0	3.8	7.0
Radix	1.8	3.5	6.3
FFT	1.8	3.5	6.1
SOR	1.8	3.3	6.8

MM is a coarse grain application that performs a multiplication operation of two square matrixes of size D. Since there is practically no synchronization between threads, MM achieved speedup of 7.8 on 8 nodes. MM spent 99.2% of its execution time doing useful computation and 0.7% waiting for remote pages on page misses.

LU is a single-writer application with coarse-grain access that performs blocked LU factorization of a dense matrix. LU sent a total of 83,826 data messages and 164,923 control messages. Data message is related to the information needed during the execution of the application, such as pages that are transmitted during the execution are counted as data messages. Control is related to the information needed for the correct work of the software DSM protocol, such as page invalidations. Compared with the C version, Java sent 28 times more control messages and 20 times more data messages. LU required almost 1 MB/s of bandwidth per CPU and achieved speedup of 7 on 8 nodes. Two components contribute to LU' slowdown: page and barrier. Page access time increased approximately 23 times, when we compare the execution on 8 nodes with the execution on 2 nodes. Page misses contributed to 3.6% of the total execution time. In the C version of LU, page misses contributed to 2.7% of the execution time. This happened because the number of page misses in Java is 19 times that of C version. Although barrier time increased less than page miss time (11%), it corresponded to 9.6% of the total execution time, against that of 6.4% in C.

Radix is a multiple-writer application with coarse-grain access that implements an integer sort algorithm. It sent a total of 1,883 data messages and 2,941 control messages. Compared with the C version, Java sent 1.9 times less control messages and 2.9 times less data message. Radix required 1.9 MB/s of bandwidth per CPU and achieved a good speedup of 6.3 on 8 nodes. Three components contributed to slow down this benchmark: page, barrier and handler. Page misses contributed to 9% of the total execution time, barrier to 6.5% and handler to 4.2%. In the C version of the algorithm, page misses contributed to 8% of the execution, handler to 12.3% and barrier to 45%. We observed that in C implementation there were 4.7 times more *diffs* created and applied than that of the Java version. The smaller number of *diffs* also had impact on the total volume of bytes transferred and on the miss rate of cache level 1. The C version transferred almost 2 times more bytes than that of Java, and its cache level 1 miss rate was almost 10 times higher than that of Java.

FFT implements the Fast Fourier Transform algorithm. Communication occurs in transpose steps, which require all-to-all thread communication. FFT is a single-writer application with fine-grained access. It sent a total of 68 MB of data, which is 4.7 times more than that of the C version. FFT required 1.8 MB/s of bandwidth per CPU and achieved speedup of 6.1 on 8 nodes. Page misses and barrier contributed to slow down this application. Page misses contributed to 15.8% of the total execution time, while barrier contributed with almost 10.7%. In the C version of the algorithm, page misses contributed to 18% of the execution and barrier to 3.8%. The miss rate on the level 1 cache is almost 7 times higher in C than in Java.

Table 4. HLRC statistics on 8 nodes: CoJVM (first figure) versus C (second figure)

Statistic	SOR	LU	Radix	FFT
Page faults	21,652 / 1,056	172,101 / 4,348	3,818 / 5,169	20,110 / 9,481
Page misses	690 / 532	81,976 / 4,135	1,842 / 958	15,263 / 3,591
Lock acquired	4 / 0	39 / 0	34 / 24	40 / 0
Lock misses	4 / 0	6 / 0	1 / 24	11 / 0
Diff created	20,654 / 0	79,338 / 0	946 / 4,529	1840 / 0
Diff applied	20,654 / 0	79,337 / 0	946 / 4,529	1840 / 0
Barrier	150 / 150	257 / 257	11 / 11	6 / 6

SOR uses the red-black successive over-relaxation method for solving partial differential equations. Communication occurs across the boundary rows between bands and is synchronized with barriers. This explains why the barrier time was responsible for 13% of execution. In the C version of SOR, barrier time was responsible for just 2.5% of the execution time. The difference between Java and C versions is due to an imbalance in computation caused by the high amount of *diffs* that Java creates. Because of its optimizations, the C version did not create *diffs*. Java created more than 20,000 *diffs*, which caused an imbalance due the deliver of write notices at the barrier. *Diff* is also responsible for 94% of the total data traffic. Data traffic in Java is 21 times more than in the C version. SOR required 1 MB/s of bandwidth per CPU and achieved speedup of 6.8 on 8 nodes. We can observe in Table 3 that the speedup obtained from 2 and 4 nodes are worse than those obtained with 4 and 8 nodes, respectively. Further investigation revealed that the larger caches were responsible for improving speedups.

The above results show that two components are the main responsible for slowing down the benchmarks: page miss and barrier time. The barrier synchronization time is mostly caused by an imbalance in execution times between processors. This imbalance seems to stem mainly from inadequate distribution of pages among nodes, although false sharing could also affect performance. This imbalanced distribution of pages among home nodes occurs because usually one thread is responsible for the initializa-tion of internal objects and fields used during computation. Indeed, the JVM Specifica-tion establishes that all fields of an object must be initialized with their initial or default values during the object's instantiation. In our implementation, however, whenever a thread writes to a page for the first time the thread's node becomes the page's home for the entire execution, according to the "first touch" rule of the HLRC protocol. This ex-plains the imbalance we observed. One immediate solution is to perform a distributed initialization, which may not be so simple. Actually, we are studying techniques to fix this problem in a transparent fashion. The imbalance in the distribution of home nodes impacts also page faults, page misses, and the creation of *diffs* - and consequently, the total amount of control and data messages that CoJVM transfers. This is evident when we compare Java against the C implementation of the same algorithm (see Table 4). SOR, LU and FFT did not create *diffs* in C, while Java created *diffs* in large amount. Radix is the only exception: CoJVM created 4.7 times less *diffs* than C. Finally, we verify the small impact of the internal CoJVM synchronization on application performance. In SOR, this overhead was equal to 2.4 ms, i.e., the cost of 4 lock misses. Lock misses occur when the thread needs a lock (due to a monitor enter operation), but the lock must be acquired remotely. FFT presented the highest overhead, with 11 locks misses. Surprisingly, although in Radix CoJVM requested more locks than that of C version, in CoJVM just 1 lock resulted in overhead, against 24 locks in C. However, in Radix lock requests did not have any impact on performance.

6 Related Work

In this section we describe some distributed Java systems with implementations based on software DSM. A detailed survey on Java for high performance computing, including systems that adopt message-passing approaches, can be found in [12].

Java/DSM [13] was the first proposal of shared-memory abstraction on top of a heterogeneous network of workstations. In Java/DSM, the heap is allocated in the shared memory area, which is created with the use of TreadMarks [14], a homeless LRC protocol, and classes read by the JVM are allocated automatically into the shared memory. In this regard, CoJVM adopts a similar to approach, but using a home-based protocol. Java/DSM seems to be discontinued, and did not report any experimental result. cJVM [15] supports the idea of single system image (SSI) using the proxy design pattern [16], in contrast to our approach that adopts a software DSM protocol. In cJVM a new object is always created in the node where the request was executed first. Every object has one master copy that is located in the node where the object is created; objects from the other nodes that access this object use a proxy. If an object is heavily accessed, the node where the master copy is located becomes potentially a bottleneck. DISK [9] adopts an update-based, object-based, multiple-writer memory consistency protocol for a distributed JVM. CoJVM differs from DISK in two aspects: a) CoJVM adopts an invalidate-based approach, and b) we currently adopt a page-based approach, although our implementation is sufficient flexible to adopt an object-based or even a word-based approach. DISK detects which objects must be shared by the protocol and uses this information to reduce consistency overheads. DISK presents the speedups of two benchmarks, however without analyzing the results. JESSICA [18] adopts a home-based, object-based, invalidation-based, multiple-writer memory consistency protocol for a distributed JVM. In JESSICA, the node that started the application, called console node, performs all the synchronization operations, which impose a severe performance degradation: for

SOR, synchronization is responsible for 68% of the execution time, and the application achieves a speedup of 3.4 on 8 nodes [18]. In CoJVM the lock ownership is distributed equally among all the nodes participating of the computation, which contributed to the better performance achieved by our environment. The modifications made by Jessica in the original JVM impose a great performance slowdown in sequential application: for SOR, this slowdown is equal to 131%, while CoJVM almost do not impose any overhead.

7 Conclusion and Ongoing Works

In this work, we introduced and evaluated CoJVM, a cooperative JVM that addresses in a novel way several performance aspects related to the implementation of DSM in Java. CoJVM complies with the Java language specification while supporting the shared memory abstraction as implemented by our customized version of HLRC, a home-based software DSM protocol. Moreover, CoJVM uses VIA as its communication protocol aiming to improve Java application performance even further.

Using several benchmarks we showed that CoJVM achieved speedups, ranging from 6.1 to 7.8 on 8 nodes. However, we believe that CoJVM can further improve application speedups. In particular, we noticed that a shared data distribution imbalance can significantly impact barrier times, page faults, page misses and the creation of *diffs* – and consequently, the total amount of control and data messages that CoJVM transfers unnecessarily. We are studying new solutions to overcome this imbalance while refining CoJVM to take advantage of the application behavior, extracted from the JVM during run-time, in order to reduce the overheads of the coherence protocol. More specifically, a specialized run-time JVM machinery is being developed to create *diffs* dynamically, to allow the use of smaller units of coherence, and to detect automatically reads and writes to the shared memory, without using the time-expensive virtual-memory protection mechanism.

References

1. Arnold, K; Gosling, J. The Java Programming Language. Addison-Wesley, 1996.
2. Lindholm, T, Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1999.
3. Lobosco, M, Amorim, C, Loques, O. A Java Environment for High-Performance Computing. 13th Symp. on Computer Architecture and High-Performance Computing, Sep 2001.
4. Zhou, Y, et alli. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. OSDI, Oct 1996.
5. Li, K, Hudak, P. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321–359, Nov 1989.
6. Keleher, P, Cox, A, Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. Int. Symp. on Computer Architecture, pp. 13–21, May 1992.
7. Rangarajan M, Iftode L. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. ALS, Oct 2000.
8. VIA. *VIA Specification, Version 1.0.* http://www.viarch.org. Accessed on Jan, 29.

9. Surdeanu, M, Moldovan, D. Design and Performance Analysis of a Distributed Java Virtual Machine. IEEE Trans. on Parallel and Distributed Systems, Vol. 13, No. 6, Jun 2002.
10. Woo, S, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. Int. Symp. on Computer Architecture, pp. 24–36, Jun 1995.
11. Performance Counter Library. http://www.fz-juelich.de/zam/PCL/. Accessed on January, 29
12. Lobosco, M, Amorim, C, Loques, O. Java for High-Performance Network-Based Computing: a Survey. Conc. and Computation: Practice and Experience: 2002(14), pp 1–31.
13. Yu, W.; Cox, A. Java/DSM: a Platform for Heterogeneous Computing. ACM Workshop on Java for Science and Engineering Computation, Jun 1997.
14. Keleher, P, et alli. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Winter Usenix Conference, pp.115–131, Jan 1994.
15. Aridor, Y, Factor, M, Teperman, A. cJVM: a Single System Image of a JVM on a Cluster. Int. Conf. on Parallel Processing, Sep 1999.
16. Gamma, E, et alli. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
17. Fang, W, et alli. Efficient Global Object Space Support for Distributed JVM on Cluster. Int. Conf. on Parallel Processing, Aug 2002.
18. Ma, M, Wang, C, Lau, F. JESSICA: Java-Enabled Single-System-Image Computing Architecture. Journal of Parallel and Distributed Computing (60), pp. 1194–1222, 2000.