A New Distributed Java Virtual Machine for Cluster Computing

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

[1] Laboratório de Computação Paralela, Programa de Engenharia de Sistemas e Computação, 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 the Cooperative Java Virtual Machine (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 memory model by enabling multiple standard JVMs to work cooperatively and transparently to support a single distributed shared-memory across the cluster's nodes. Mostly important, CoJVM requires no change to applications written in standard Java. Our experimental results using several Java benchmarks show that CoJVM performance is respectable with speed-ups ranging from 6.1 to 7.8 for a 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 concurrent 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 as proposed by Hoare [2].

The development of parallel applications using Java's concurrent programming is restricted to shared-memory computers, which are often expensive and do not scale easily. A compromise 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 [3]. Yet, very few reported good performance results [11] and presented detailed performance analysis [23].

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) [4]. CoJVM's main objective is to speed up Java applications executing on homogeneous computer clusters, our target platform. CoJVM relies on two key features to improve application performance: 1) the HLRC software Distributed Shared Memory (DSM) protocol [5] and 2) a new instrumentation mechanism [4] to the JVM that enables new latency-tolerance techniques to exploit the application run-time behavior. Mostly important, 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 Java Virtual Machine (JVM).

In this work, we evaluate CoJVM performance for six parallel applications: Matrix Multiplication (MM), Successive Over Relaxation (SOR), Crypt, 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 even without any change in the syntax or in the semantic of the language; c) to present detailed performance analysis of CoJVM for six parallel benchmarks, and d) to

show that current multithreading Java benchmarks from the Java Grande Forum suite need to be restructured for cluster computing.

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 six Java parallel applications executed under CoJVM. In section 6, we describe related works that also implement the notion of shared memory in distributed Java environments. 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 virtual memory protection mechanism, as proposed by Li [6]. 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 lead to unnecessary communication traffic; and (b) high OS costs of treating page faults and crossing memory protection boundaries.

Several relaxed memory models, such as Lazy Release Consistency (LRC) [7], have been proposed 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, modifications to the page, called *diffs*, can be obtained at any time by comparing current copy with its twin. LRC imposes to the programmer the use of two explicit synchronization 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 acquirer will receive from the last acquirer all the write-notices, which correspond to modifications made to the pages that the acquirer has not seen according to the happen-before-1 partial order [7]. HLRC [8] introduced the concept of home node, in which each node is responsible for maintain an up-to-date copy of its pages. The acquirer then request copies of modified pages to the home nodes. At release points, *diffs* are computed and sent to the page's home node, which reduces memory consumption in home-based DSM protocols and contributes to the scalability of the HLRC protocol [8].

3 Java

The Java language specification allows for a software Java DSM implementation. This section describes the necessary Java concepts we applied to the CoJVM design.

3.1 Multithreading and Synchronization in Java

In Java, threads programming is simple since it provides a native 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 fashion according to the monitor paradigm [2].

3.2 Memory Model

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 [3]. The main memory stores all program variables and is shared by the JVM threads (refer to Figure 1). 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.

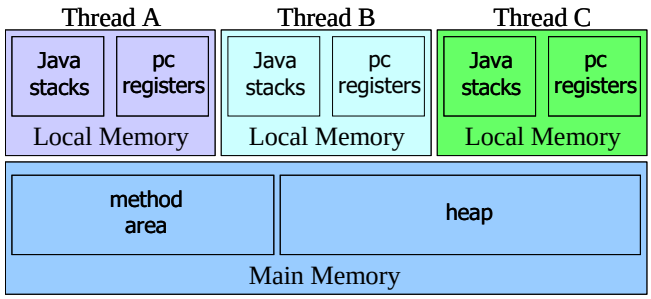

Fig. 1. The internal architecture of the Java Virtual Machine's Memory

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. The primitives support blocks of code declared as synchronized. 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 [4] is a distributed shared memory (DSM) implementation of the standard JVM and was designed carefully 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 on the Java Virtual Machine [3]. Therefore in CoJVM all declared objects are implicitly and automatically allocated into the Java heap, which is implemented in the distributed shared-memory space. Our DSM implementation uses the HLRC (Home-based, Lazy Release Consistency) protocol for two main reasons. First, it tends to consume less memory and its scalable performance is competitive with that of homeless LRC implementations [8]. Second, the HLRC implementation [9] already supports the VI Architecture (VIA) [10], a high-performance user-level communication protocol. In [11], it has been showed 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, as we plan to support multiple units of granularity in accord with Java's primitive types in the next CoJVM version.

CoJVM benefits from the fact that Java already provides synchronization primitives. The synchronized reserved word permits the definition of blocks or methods that must be executed under mutual exclusion. The wait method forces an object to wait until a state is reached. The notify and notifyAll methods notify one or all objects, respectively, that some condition was changed. The programmer with the use of these primitives can easily define a barrier or other synchronization constructs. CoJVM supports only Java standard features, and adds no extra synchronization primitive.

Since the declaration and synchronization of objects in the distributed shared memory follow the Java model and no extra synchronization primitive is added in our environment, a standard concurrent application can run in our environment without any code change. Threads created by the application are automatically moved to a remote host, and communication among them are treated following the language specification in a transparent way.

The main difference between CoJVM and current DSM-based Java implementations is that CoJVM takes advantage of the run-time application behavior, extracted from JVM, to reduce the overheads of the

coherence protocol. More specifically, a specialized run-time JVM machinery (data and support mechanism) is used to create *diffs* dynamically, to allow the use of smaller data coherence units, and to detect automatically reads and writes to the shared memory, without using the expensive virtual-memory protection mechanism. However, in the experiments described in this paper we do not use the internal state extracted from the JVM to optimize the DSM protocol.

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.

To evaluate the performance of CoJVM, we attempted to use the Java Grande Forum (JGF) multithreading benchmark suite [12], but only the Crypt benchmark, which is an implementation of the IDEA algorithm, performed well. Other two benchmarks, namely SOR and LU, produced severe performance degradation, although they achieved good speedup when running on a SMP machine. We identified problems in data distribution and task parallelism, which explained the poor performance of both applications for cluster computing. In LU, for example, there is only a small portion of the code that is parallel, and a large sequential block that could be restructured for running in parallel. Due these problems, we decided to port four benchmarks from the SPLASH-2 benchmark suite [13]: SOR, LU, FFT and Radix, and developed one, MM, a Matrix Multiplication algorithm.

The sequential execution time for each application is 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%.

Table 1. Sequential times of applications

Program	Program Size	Seq. Time (s)
LU	2048x2048	2,345.88
Crypt	5000000	304.43
MM	1000x1000	485.77
Radix	8388608 Keys	24.27
FFT	4194304 Complex Doubles	227.21
SOR	2500x2500	304.75

To quantify the overhead that CoJVM imposes due to its modifications of the standard JVM, we instrumented both machines with the performance counter library PCL [14], which collects 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 standard Java and on CoJVM. The table also presents the percentage of load/store instructions and data cache level 1 miss rates.

Table 2. Performance of sequential applications

Counter	JVM / CoJVM			
	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

Table 2 shows that the percentage of load/store instructions does not change for Radix and FFT, but increases for LU and decreases slightly for SOR. The increase in the number of memory accesses in LU helps to explain why CoJVM miss rate is 5 times higher than JVM. Notice that increasing the amount of memory accesses, the probability of cache misses increases. Also, 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 the number of messages and the amount of bytes the applications required when running

on a 8-node cluster as well as the average message size (in kilobytes) and the average bandwidth per processor. The table shows two distinct categories of messages: data and control. Data message is related to the information needed during the execution of the application. For example, *diffs* and 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. To compute the average bandwidth per processor we added the total number of bytes that were transferred (data and control) and divided the result by the total execution time. We observe that the benchmarks have very different characteristics. MM sent the smallest number of messages and bytes, while LU transferred the largest amount of both. Although Radix sent the second smallest amount of bytes, its low execution time turned it the application that required the highest bandwidth.

Table 3. Message and data for 8 processors

Program	Messages		Transfers (MB)		Medium Size (KB)		Bw per CPU (MB/s)
	Data	Control	Data	Control	Data	Control	
LU	83,826	164,923	323.69	8.65	3.95	0.05	0.99
Crypt	4,018	4,022	15.69	0.29	4.00	0.07	0.40
MM	1,186	1,186	4.62	0.08	3.99	0.07	0.08
Radix	1,883	2,941	7.21	0.14	3.92	0.05	1.92
FFT	17,095	17,212	66.63	1.30	3.99	0.08	1.83
SOR	21,215	23,454	44.50	0.70	2.15	0.03	1.00

Table 4 shows application speedups. We can observe that two applications, MM and Crypt, achieved almost linear speedups 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 4. Speedups

Program	2 Nodes	4 Nodes	8 Nodes
LU	2.0	3.8	7.0
Crypt	1.9	3.8	7.6
MM	2.0	4.0	7.8
Radix	1.8	3.5	6.3
FFT	1.8	3.5	6.1
SOR	1.8	3.3	6.8

Figure 2 shows how execution time is spent on the benchmarks. The execution time is broken down into six distinct components: computation, page, lock, barrier, overhead, and handler time. Computation indicates the time spent in useful computation. Page indicates the amount of time spent in fetching pages from remote home nodes on page misses. Lock is the time spent in getting the lock from its current owner. Barrier is the time spent waiting for barrier messages from other nodes, at the barrier. Overhead time is the time spent performing protocol actions. Handler time is the time spent inside the handler, a separate thread used to process requests from remote nodes.

5.1 LU

LU performs blocked LU factorization of a dense matrix. The matrix is decomposed in contiguous blocks that are distributed to nodes in contiguous chunks.

LU is a single-writer application with coarse-grain access. LU sent a total of 83,826 data messages and 164,923 control messages. 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. In Figure 2a, we can observe that 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.

5.2 Crypt

Crypt performs the IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. This algorithm involves two main loops, whose iterations are independent and divided among the threads in a block fashion. This explains the near-linear speedup at 7.6 on 8 nodes achieved by IDEA. Page misses contributed to 1.7% of the total execution time, and explains why the benchmark does not achieve linear speedup.

5.3 MM

MM is a coarse grain application that performs a multiplication operation of two square matrixes of size D. This algorithm involves three main loops, whose iterations are independent and are divided between the threads in a block fashion. 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.

5.4 Radix

Radix implements an integer sort algorithm. Keys are divided among threads, and in each program iteration, a thread passes over its assigned keys and generates a local histogram of values, which is accumulated into a global histogram. The global histogram is used to permute the keys for the next iteration.

Radix is a multiple-writer application with coarse-grain access. 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. In Figure 2b, we can observe that 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.

5.5 FFT

The 1D-FFT data set consists of n complex data points to be transformed, and another n complex data points referred to as the roots of unity. Both sets of data are organized as matrices, which are partitioned in rows and distributed among threads. 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. As we can observe in Figure 2c, 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.

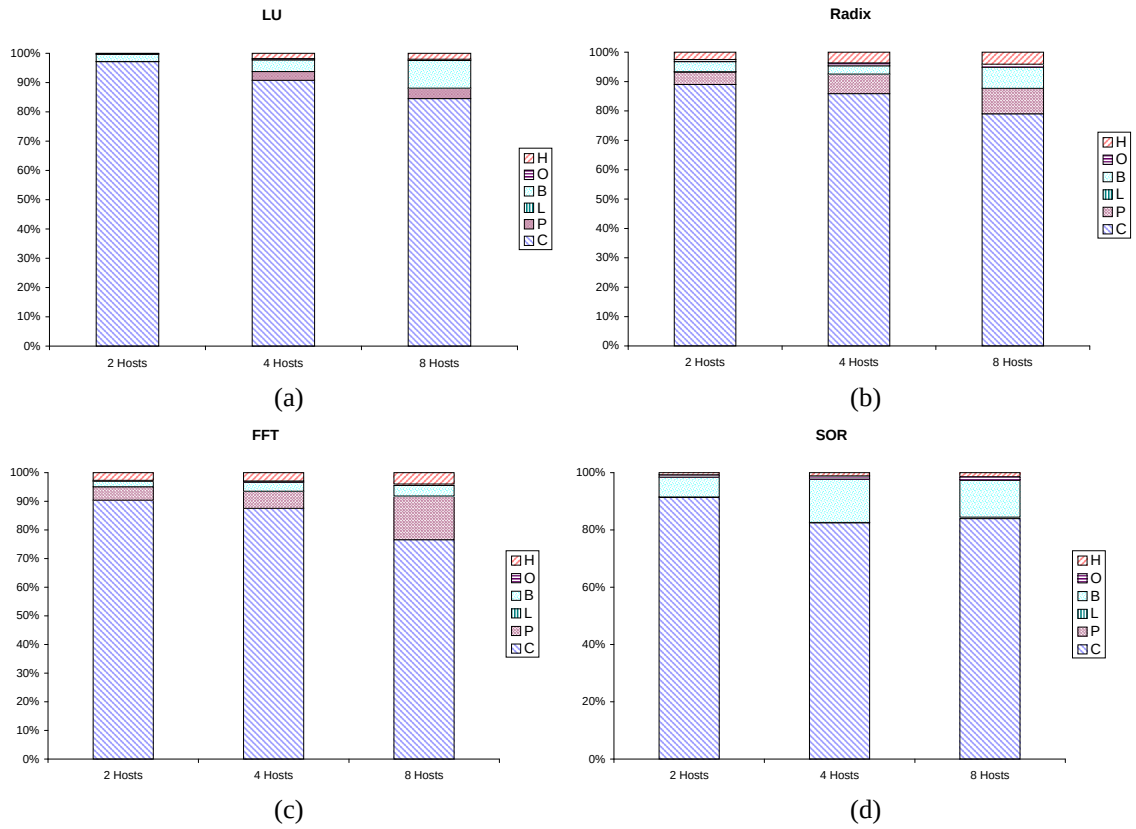

Fig. 2. Execution time breakdown – C: Computation, P: Page, L: Lock, B: Barrier, O: Overhead, H: Handle

5.6 SOR

SOR uses the red-black successive over-relaxation method for solving partial differential equations. The black and red arrays are partitioned into roughly equal size bands of rows, which are distributed among the nodes. 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 (Figure 2d). 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 4 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.

5.7 Analysis

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. The problem stems from the imbalance in the distribution of pages among nodes. This occurs because usually one thread is responsible for the initialization of internal objects and fields used during computation. Indeed, the Java Virtual Machine Specification establishes that all fields of an object must be initialized with their initial or default values during the instantiation. In our implementation, however, occurred that when a thread writes to a page the thread's node becomes the page's home according to the HLRC protocol, which led to 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 5). Notice that the C version is mature and stable, besides its data distribution is very optimized. 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, looking at Table 5 and Figure 2 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 (see Figure 2b).

Table 5. HLRC statistics on 8 nodes

Statistic	CoJVM / C			
	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

6 Related Work

In this section we describe some distributed Java systems whose implementation is based on software DSM. A detailed survey on Java for high performance computing, including systems that adopt message-passing approaches, can be found in [15].

Java/DSM [16] 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 [17], a homeless lazy release 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 [18] supports the idea of single system image (SSI). To implement the SSI abstraction, cJVM uses the proxy design pattern [19], 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. cJVM modified the semantics of the new *opcode*, allowing the creation of threads in remote nodes. CoJVM does not modify the semantics nor the syntax of any Java *opcode*.

DISK [11] 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, matrix multiplication and traveling salesman problem, however without analyzing the results.

Kaffemik [20] uses the Scalable Coherent Interface (SCI)[21] hardware support to provided the shared memory abstraction on a distributed JVM. CoJVM does not rely on special hardware support to provide the shared-memory abstraction. Like CoJVM, each node of the cluster executes an instance of the JVM. Instances collaborate to execute a parallel job. However, unlike CoJVM, Kaffemik does not support replication of objects on the instances of the JVM, and there is exactly one copy of each object, which can

degrade the performance of application. Kaffemik presents the speedups of one benchmark, ray tracer, on just three nodes. So we are not able to conclude if the performance of applications on Kaffemik scales with the increase in number of nodes.

JESSICA [22] adopts a home-based, multiple-writer memory consistency protocol for a distributed JVM. It relies on objects as the unit of coherence and uses invalidation to maintain the coherence among objects. JESSICA uses a protocol similar to Lazy Release Consistence. 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 [23]. 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.

MultiJav [24] implements the distributed shared-memory (DSM) model into Java by modifying the JVM, but using standard Java concurrency constructs, thus avoiding changing the language definition. The approach is similar to ours. However, sharing is object-based in MultiJav, while currently we adopt a page-based implementation. One of the main objectives of MultiJav is to maintain the portability of the Java language, allowing its use in heterogeneous hardware platforms. However, the author neither estimate the effort required to implement the support mechanisms in each target architectures nor measure its performance impacts. CoJVM does not address this issue, assuming a homogeneous cluster environment. MultiJav runtime system, through an analysis of the load/store instructions of the *bytecode* being executed, can automatically detect which objects should be shared, in order to guarantee consistency. However, no experimental result is presented to measure the cost of such analysis, which may impose performance penalties. All objects declared in CoJVM are automatically shared, as stated in the Java language definition. MultiJav seems to be discontinued, and did not report any experimental result.

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 distributed shared memory 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 good 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 imbalance during data initialization can impact significantly 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 solutions to overcome this problem.

We also attempted unsuccessfully to run two benchmarks from the Java Grande Forum multithreaded suite. While all benchmarks tend to perform well on SMPs, SOR and LU presented poor performance on clusters due to problems in data distribution and low parallelism.

Currently we are refining CoJVM to take advantage of the run-time application behavior, which can be extracted from the JVM, to reduce the overheads of the coherence protocol. More specifically, a specialized run-time JVM machinery (data and support mechanisms) is being written 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. We also plan to port CoJVM to the last version available of the Java platform and to develop a multithreading TPC-W [25] benchmark to evaluate the performance of CoJVM on commercial applications.

References

1 Arnold, K; Gosling, J. The Java Programming Language. First Edition. Addison-Wesley, 1996.
2 Hoare, C. Monitors: An Operating System Structuring Concept. Communications of the ACM, 12(10), Oct 1974.

3 Lindholm, T, Yellin, F. The Java Virtual Machine Specification. Second edition. Addison-Wesley, 1999.
4 Lobosco, M, Amorim, C, Loques, O. A Java Environment for High-Performance Computing. 13th SBAC-PAD, Sept 2001.
5 Zhou, Y, Iftode, L, Li, K. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, Oct 1996.
6 Li, K, Hudak, P. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321--359, Nov 1989.
7 Keleher, P, Cox, A, Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. Proceedings of the Nineteenth International Symposium on Computer Architecture, pp. 13-21, May 1992.
8 Zhou, Y, Iftode, L, Li, K. Performance Evaluation of Two Home-based Lazy Release Consistency Protoco ls for Shared Virtual Memory Systems. Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, Oct 1996.
9 Rangarajan M, Iftode L. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. Proceedings of the 4th Annual Linux Showcase and Conference, Oct 2000.
10 Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0*. http://www.viarch.org. Accessed on Jan, 29.
11 Surdeanu, M, Moldovan, D. Design and Performance Analysis of a Distributed Java Virtual Machine. IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 6, June 2002.
12 Java Grande Forum. http://www.javagrande.org. Accessed on January, 29.
13 Woo, S, et alli. The SPLASH-2 Programs: Characterization and Methodological Considerations. Proceedings of the 22nd International Symposium on Computer Architecture, pp. 24-36, June 1995.
14 Performance Counter Library. http://www.fz-juelich.de/zam/PCL/. Accessed on January, 29
15 Lobosco, M, Amorim, C, Loques, O. Java for High-Performance Network-Based Computing: a Survey. Concurrency and Computation: Practice and Experience: 2002(14), pp 1-31.
16 Yu, W.; Cox, A. Java/DSM: a Platform for Heterogeneous Computing. ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.
17 Keleher, P, Dwarkadas, A, Cox, A, Zwaenepoel, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Proceedings of the 1994 Winter Usenix Conference, pp.115-131, January 1994.
18 Aridor, Y, Factor, M, Teperman, A. cJVM: a Single System Image of a JVM on a Cluster. International Conference on Parallel Processing 99 (ICPP 99), September 1999.
19 Gamma, E, et alli. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
20 Anderson, J, et alli. Kaffemik – A Distributed JVM on a Single Address Space Architecture. 4th International Conference on SCI-based Technology and Research. Oct 2001.
21 IEEE. IEEE Standard for Scalable Coherent Interface. IEEE standard 1596-1992, Aug 1993.
22 Fang, W, et alli. Efficient Global Object Space Support for Distributed JVM on Cluster. International Conference on Parallel Processing, Aug 2002.
23 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.
24 Chen, X.; Allan, V. MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. The 1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98), July 1998.
25 Transaction Processing Performance Council. http://www.tpc.org/. Accessed on January, 29.