Compiling multithreaded Java bytecode for distributed execution

Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark Macbeth, Keith Mcguigan, Raymond Namyst

Compiling Multithreaded Java Bytecode for Distributed Execution

Gabriel Antoniu[1], Luc Bougé[1], Philip Hatcher[2], Mark MacBeth[2⋆], Keith McGuigan[2], and Raymond Namyst[1]

[1] LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.
Raymond.Namyst@ens-lyon.fr
[2] Dept. Computer Science, Univ. New Hampshire, Durham, NH 03824, USA.
Philip.Hatcher@unh.edu

Abstract. Our work combines Java compilation to native code with a run-time library that executes Java threads in a distributed-memory environment. This allows a Java programmer to view a cluster of processors as executing a *single* Java virtual machine. The separate processors are simply resources for executing Java threads with true concurrency and the run-time system provides the illusion of a shared memory on top of the private memories of the processors. The environment we present is available on top of several UNIX systems and can use a large variety of network protocols thanks to the high portability of its run-time system. To evaluate our approach, we compare serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. All measurements have been carried out on two platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

1 Introduction

The Java programming language is an attractive vehicle for constructing parallel programs to execute on clusters of computers. The Java language design reflects two emerging trends in parallel computing: the widespread acceptance of both a threads programming model and the use of a distributed-shared memory (DSM). While many researchers have endeavored to build Java-based tools for parallel programming, we think most people have failed to appreciate the possibilities inherent in Java's use of threads and a "relaxed" memory model.

There are a large number of parallel Java efforts that connect multiple Java virtual machines by utilizing Java's remote-method-invocation facility (e.g., [4, 5, 10, 13]) or by grafting an existing message-passing library (e.g., [7, 8]) onto Java. In our work we view a cluster as executing a single Java virtual machine. The separate nodes of the cluster are hidden from the programmer and are simply resources for executing Java threads with true concurrency. The separate

⋆ Mark MacBeth is currently affiliated with Sanders, A Lockheed Martin Company, PTP02-D001, P.O. Box 868, Nashua, NH, USA.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1039–1052, 2000.

memories of the nodes are also hidden from the programmer and our implementation must support the illusion of a shared memory within the context of the Java memory model, which is "relaxed" in that it does not require sequential consistency.

Our approach is most closely related to efforts to implement Java interpreters on top of a distributed shared memory [2, 6, 17]. However, we are interested in computationally intensive programs that can exploit parallel hardware. We expect the cost of compiling to native code will be recovered many times over in the course of executing such programs. Therefore we focus on combining Java compilation with support for executing Java threads in a distributed-memory environment.

Our work is done in the context of the Hyperion environment for the high-performance execution of Java programs. Hyperion was developed at the University of New Hampshire and comprises a Java-bytecode-to-C translator and a run-time library for the distributed execution of Java threads. Hyperion has been built using the PM2 distributed, multithreaded run-time system from the École Normale Supérieure de Lyon [12]. As well as providing lightweight threads and efficient inter-node communication, PM2 provides a generic distributed-shared-memory layer, DSM-PM2 [1]. Another important advantage of PM2 is its high portability on several UNIX platforms and on a large variety of network protocols (BIP, SCI, VIA, MPI, PVM, TCP). Thanks to this feature, Java programs compiled by Hyperion can be executed with true parallelism in all these environments.

In this paper we describe the overall design of the Hyperion system, the strategy followed for the implementation of Hyperion using PM2, and a preliminary evaluation of Hyperion/PM2 by comparing serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. The evaluation is performed on two different platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

2 The Hyperion System

2.1 Compiling Java

Our vision is that programmers will develop Java programs using the workstations on their desks and then submit the programs for production runs to a "high-performance Java execution server" that appears as a resource on the network. Instead of the conventional Java paradigm of pulling bytecode back to their workstation for execution, programmers will push bytecode to the high-performance server for remote execution. Upon arrival at the server the bytecode is translated for native execution on the processors of the server. We utilize our own Java-bytecode-to-C compiler (`java2c`) for this task and then leverage the native C compiler for the translation to machine code.

As an aside, note that the security issues surrounding "pushing" or "pulling" bytecodes can be handled differently. When pulling bytecodes, users want to

bring applications from potentially untrusted locations on the network. The Java features for bytecode validation can be very useful in this context. In contrast, when "pushing" bytecodes to a high-performance Java server, conventional security methods might be employed, such as only accepting programs from trusted users. However, the Java security features could still be useful if one wanted to support an "open" Java server, accepting programs from untrusted users.

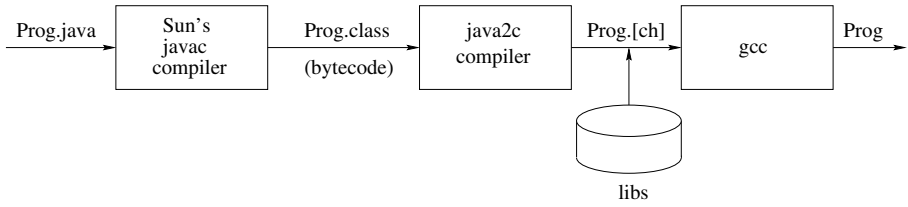

Fig. 1. Compiling Java programs with Hyperion

Code generation in `java2c` is straightforward (see Figure 1). Each virtual machine instruction is translated directly into a separate C statement, similar to the approaches taken in the Harissa [11] or Toba [14] compilers. As a result of this method, we rely on the C compiler to remove all the extraneous temporary variables created along the way. Currently, `java2c` supports all non-wide format instructions as well as exception handling.

The `java2c` compiler also includes an optimizer for improving the performance of object references with respect to the distributed-shared memory. For example, if an object is referenced on each iteration of a loop, the optimizer will lift out of the loop the code for obtaining a locally cached copy of the object. Inside the loop, therefore, the object can be directly accessed with low overhead via a simple pointer. This optimization needs to be supported by both compiler analysis and run-time support to ensure that the local cache will not be flushed for the duration of the loop.

2.2 The Hyperion Run-Time System Design

To build a user program, user class files are compiled (first by Hyperion's `java2c` and then the generated C code by a C compiler) and linked with the Hyperion run-time library and with the necessary external libraries. The Hyperion run-time system is structured as a collection of modules that interact with one another (see Figure 2). We now present the main ones.

Java API Support. Hyperion currently uses the Sun Microsystems JDK 1.1 as the basis for its Java API support. Classes in the Java API that do not include native methods can simply be compiled by `java2c`. However, classes with native methods need to have those native methods written by hand to fit the Hyperion

design. Unfortunately, the Sun JDK 1.1 has a large number of native methods scattered throughout the API classes. To date, we have only implemented a small number of these native methods and therefore our support for the full API is limited. We hope that further releases of Java 2 (e.g., Sun JDK 1.2) will be more amenable to being compiled by `java2c`.

Threads Subsystem. The threads module provides support for lightweight threads, on top of which Java threads can be implemented. This support obviously includes thread creation and thread synchronization. For portability reasons, we model the interface to this subsystem on the core functions provided by POSIX threads. Thread migration is also available, thanks to PM2's support. We plan to use this feature in future investigations of dynamic and transparent application load balancing.

Communication Subsystem. The communication module supports the transmission of messages between the nodes of a cluster. The interface is based upon message handlers being asynchronously invoked on the receiving end. This type of interface is mandatory since most communications, either one-way or round-trip, must occur without any explicit contribution of the remote node: incoming requests are handled by a special daemon thread which runs concurrently with the application threads. For example, in our implementation of the Java memory model, one node of a cluster can asynchronously request data from another node.

Memory Subsystem. The Java memory model [9] allows threads to keep locally cached copies of objects. Consistency is provided by requiring that a thread's object cache be flushed upon entry to a monitor and that local modifications made to cached objects be transmitted to the central memory when a thread exits a monitor. Table 1 provides the key primitives of the Hyperion memory subsystem that are used to provide Java consistency. The DSM environment on top of which they are built is required to provide direct support for their implementation. This condition is fulfilled by the API of the DSM layer of PM2 (see Section 3.2 for additional details).

`loadIntoCache`	Load an object into the cache
`invalidateCache`	Invalidate all entries in the cache
`updateMainMemory`	Update memory with modifications made to objects in the cache
`get`	Retrieve a field from an object previously loaded into the cache
`put`	Modify a field in an object previously loaded into the cache

Table 1. Key DSM primitives

Hyperion's memory module also includes mechanisms for object allocation, garbage collection and distributed synchronization. Java monitors and the associated wait/notify methods are supported by attaching mutexes and condition variables from the Hyperion threads module to the Java objects managed by the Hyperion memory layer.

Load Balancer. The load balancer is responsible for choosing the most appropriate node on which to place a newly created thread. The current strategy is rather simple: threads are assigned to nodes in a round-robin fashion. We use a distributed algorithm, with each node using round-robin placement of its locally created threads, independently of the other nodes. More complex load balancing strategies based on dynamic thread migration and on the interaction between thread migration and the memory consistency mechanisms are currently under development.

3 Hyperion/PM2 Implementation Details

The current implementation of Hyperion is based on the PM2 distributed multithreaded environment (Figure 2). PM2's programming interface allows threads to be created locally and remotely and to communicate through RPCs (Remote Procedure Calls). PM2 also provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such functionality is typically useful to implement dynamic load balancing policies. The interactions between thread migration and data sharing are handled through a *distributed shared memory* facility: the *DSM-PM2* [1] layer.

Most Hyperion run-time primitives in the threads, communication and shared memory subsystems are implemented by directly mapping onto the corresponding PM2 functions.

3.1 Threads and Communication

Threads Subsystem. The threads component of Hyperion is a very thin layer that interfaces to Marcel (PM2's thread library). Marcel is an efficient, user-level, POSIX-like thread package featuring thread migration. Most of the functions in Marcel's API provide the same syntax and semantics as the corresponding POSIX Threads functions. However, it is important to note that the Hyperion thread component uses the PM2 thread component through the PM2 API and does not access the thread component directly, as would be typical when using a classical *Pthreads*-compliant package. PM2 implements a careful integration of multithreading and communication that actually required several modifications of the thread management functions (e.g., thread creation). Thus, it would be inefficient (and even dangerous) to bypass the PM2 API by using the underlying thread package directly.

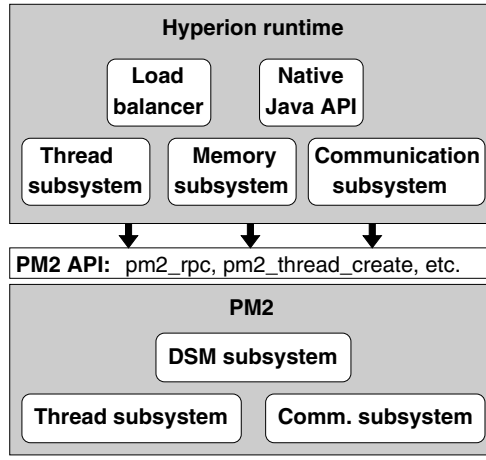

Fig. 2. Overview of the Hyperion software architecture

Communication Subsystem. The communication component of Hyperion is implemented using PM2 remote procedure calls (RPCs), which allow PM2 threads to invoke the remote execution of user-defined services (i.e., functions). On the remote node, PM2 RPC invocations can either be handled by a pre-existing thread or they can involve the creation of a new thread. This latter functionality allows us to easily implement Hyperion's communication subsystem. PM2 utilizes a generic communication package [3] that provides an efficient interface to a wide-variety of high-performance communication libraries, including low-level ones. The following network protocols are currently supported: BIP (Myrinet), SISCI (SCI), VIA, MPI, PVM and TCP.

3.2 Memory Management

The memory management primitives described in Table 1 are implemented on top of PM2's distributed-shared-memory layer, DSM-PM2 [1]. DSM-PM2 has been designed to be generic enough to support multiple consistency models. Sequential consistency and Java consistency are currently available. Moreover, for a given consistency model, alternative protocols (based on page migration and/or on thread migration) are provided. Also, new consistency models can be easily implemented using the existing generic DSM-PM2 library routines.

DSM-PM2 is structured in layers. At the high level, a *DSM protocol policy* layer is responsible for implementing consistency models out of a subset of the available library routines and for associating each application data with its own consistency model. The library routines (used to bring a copy of a page to a thread, to migrate a thread to a page, to invalidate all copies of a page, etc.) are grouped in the lower-level *DSM protocol library* layer. Finally, these library routines are built on top of two base components: the *DSM page manager* and the *DSM communication module*. The *DSM page manager* is essentially dedicated

to the low-level management of memory pages. It implements a distributed table containing page ownership information and maintains the appropriate access rights on each node. The *DSM communication module* is responsible for providing elementary communication mechanisms, such as delivering requests for page copies, sending pages, and invalidating pages.

The DSM-PM2 user has three alternatives that may be utilized according to the user's specific needs: (1) use a built-in protocol, (2) build a new protocol out of a subset of library routines, or (3) write new protocols using the API of the *DSM page manager* and *DSM communication module* (for more elaborate features not implemented by the library routines). The Hyperion DSM primitives (`loadIntoCache`, `updateMainMemory`, `invalidateCache`, `get` and `put`) have been implemented using this latter approach.

Object replication: main memory and caches. To implement the concept of *main memory* specified by the Java model, the run-time system associates a *home node* to each object. The home node is in charge of managing the reference copy. Initially, the objects are stored on their home nodes. They can be replicated if accessed on other nodes. Note that at most one copy of an object may exist on a node and this copy is shared by all the threads running on that node. Thus, we avoid wasting memory by associating caches to nodes rather than to threads.

Access detection and modification recording. Hyperion uses specific access primitives to shared data (`get` and `put`), which allows us to use explicit checks to detect if an object is present (i.e., has a copy) on the local node. If the object is present, it is directly accessed, else the page(s) containing the object is locally cached. Thanks to the access primitives, the modifications can be recorded at the moment when they are carried out. For this purpose, a bitmap is created on a node when a copy of the page is received. The `put` primitive uses it to record all writes to the object, using *object-field granularity*. All local modifications are sent to the home node of the page by the `updateMainMemory` primitive.

Implementing objects on top of pages. Java objects are implemented on top of DSM-PM2 pages. If an object spans several pages, all the pages are cached locally when the object is loaded. Consequently, loading an object into the local cache may generate prefetching, since all objects on the corresponding page(s) are actually brought to the current node. Similarly, when updating the master copy of an object, other objects located on the same page will get their master copy updated. This implementation has been carefully designed to be fully compliant with Java consistency [9].

Object ownership. Our implementation allocates all Java objects within the section of memory controlled by DSM-PM2. We align Java objects on 2^k-byte boundaries. An object reference is basically the address of the object, but now we can use the bottom k bits to store the node number of the owner of the object. (k can be adjusted to accommodate larger number of nodes, at the expense of increasing internal fragmentation.) This allows us to do an efficient ownership test in the `loadIntoCache` primitive with a bitwise

AND, a subtract, and a test against zero. If the ownership test fails, then the DSM-PM2 page table is consulted to see if the page containing the object is locally cached. In addition, each object is given a standard header and one of the bits of the header is used to indicate if the object is a cached copy or not. This bit is used by the `put` primitive to quickly determine whether the page is locally owned or not. If not, the modification needs to be recorded in the bitmap associated with the DSM-PM2 page holding the cached copy of the object.

4 Performance Evaluation: Minimal-Cost Map-Coloring

4.1 Experimental Conditions and Benchmark Programs

We have implemented branch-and-bound solutions to the minimal-cost map-coloring problem, using serial C, serial Java, and multithreaded Java. These programs have first been run on a eight-node cluster of 200 MHz Pentium Pro processors, running Linux 2.2, interconnected by a Myrinet network and using MPI implemented on top of the BIP protocol [15]. We have also executed the programs *without any modification* on a four-node cluster of 450 MHz Pentium II processors running Linux 2.2, interconnected by a SCI network using the SISCI protocol.

The serial C program is compiled using the GNU C compiler, version 2.7.2.3 with `-O6` optimization, and runs "natively" as a normal Linux executable. The Java programs are translated to C by Hyperion's `java2c` compiler, the generated C code is also compiled by GNU C with `-O6` optimization, and the resulting object files are linked with the Hyperion/PM2 run-time system.

The two serial programs use identical algorithms, based upon storing the search states in a priority queue. The queue first gives priority to states in the bottom half of the search tree and then secondly sorts by bound value. (Giving priority to the states in the bottom half of the search tree drives the search to find solutions more quickly, which in turn allows the search space to be more efficiently pruned.)

The parallel program does an initial, single-threaded, breadth-first expansion of the search tree to generate sixty-four search states. Sixty-four threads are then started and each one is given a single state to expand. Each thread keeps its own priority queue, using the same search strategy as employed by the serial programs. The best current solution is stored in a single location, protected by a Java monitor. All threads poll this location at regular intervals in order to effectively prune their search space.

Maintaining a constant number of threads across executions on different size clusters helps to keep fairly constant the aggregate amount of work performed across the benchmarking runs. However, the pattern of interaction of the threads (via the detection of solutions) does vary and thus the work performed also varies slightly across different runs.

All programs use a pre-allocated pool of search-state data objects. This avoids making a large number of calls to either the C storage allocation primitives

(`malloc/free`) or utilizing the Java garbage collector. (Our distributed Java garbage collector is still under development.) If the pool is exhausted, the search mechanism switches to a depth-first strategy until the pool is replenished.

For benchmarking, we have solved the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Assigning sixty-four threads to this problem in the manner described above, and using Hyperion's round-robin assignment of threads to nodes, is reasonably effective at evenly spreading the number of state expansions performed around a cluster, if the number of nodes divides evenly into the number of threads. (In the future we plan to investigate dynamic and transparent load balancing approaches that utilized the thread migration features of PM2.)

4.2 Overhead of Hyperion/PM2 vs. Hand-Written C Code

First, we compare the performance of the serial programs on a *single* 450 MHz Pentium II processor running Linux 2.2. Both the hand-written C program and the C code generated by `java2c` were compiled to native Pentium II instructions using `gcc2.7.2.3` with option `-O6`. Execution times are given in seconds.

Hand-written C	63
Java via Hyperion/PM2	324
Java via Hyperion/PM2, in-line DSM checks disabled	168
Java via Hyperion/PM2, array bound checks also disabled	98

We consider this a "hard" comparison because we are comparing against hand-written, optimized C code and because the amount of straight-line computation is minimal. This application features a large amount of object manipulation (inserting to and retrieving from the priority queue; allocating new states from the pool and returning "dead-end" states to the pool) with a relatively small amount of computation involved in expanding and evaluating a single state. In fact, the top two lines in the table demonstrate that the *original* Hyperion/PM2 execution is roughly five times slower than the execution of the hand-written C program.

The bottom two lines in the table help explain the current overheads in the Hyperion/PM2 implementation of the Java code and represent cumulative improvements to the performance of the program. In the third line of the table, the Java code is executed on a single node with the *in-line checks* disabled: in-line checks are used by Hyperion to test for the presence or the absence of a given Java object at the local node in the distributed implementation; as there is only one node at work in the case at stake, they are always satisfied. In the fourth line of the table, the array bound checks are additionally disabled. This last version can be considered as the closest to the hand-written C code. It is only 55% slower. A comparison with hand-written C++ code would probably be more fair to Hyperion, and would probably result in an even lower gap.

We can draw two conclusions from these figures. First, the in-line checks used to implement the Hyperion DSM primitives (e.g., `loadIntoCache`, `get` and `put`)

are very expensive for this application. By disabling these checks in the C code generated by `java2c`, we save nearly 50% of the execution time. (This emphasizes the map-coloring application's heavy use of object manipulation and light use of integer or floating-point calculations.) For this application it may be better to utilize a DSM-implementation technique that relies on page-fault detection rather than in-line tests for locality. This can be easily done within the context of DSM-PM2's generic support and we are currently evaluating this alternative, i.e., *in-line* vs. *page-fault* checks, with an expanded set of applications.

Second, the cost of the array-bounds check in the Java array-index operation, at least in the Hyperion implementation, is also quite significant. We implement the bounds check by an explicit test in the generated C code. In the map-coloring application, arrays are used to implement the priority queues and in the representation of search states. In both cases the actual index calculations are straightforward and would be amenable to optimization by a compiler that supported the guaranteed safe removal of such checks. Such an optimization could be implemented in `java2c`. Alternatively this optimization might be done by analysis of the bytecode prior to the execution of `java2c`. Or, the optimization could be performed on the generated C code. We plan to further investigate these alternatives in the future.

4.3 Performance of the Multithreaded Version

Next, we provide the performance of the multithreaded version of the Java program on the two clusters described in Section 4.1. Parallelizability results are presented in Figure 3: the multi-node execution times are compared to the single-node execution time of the same multithreaded Java program run with 64 threads.

On the 200 MHz Pentium Pro cluster using MPI-BIP/Myrinet, the execution time decreases from 638 s for a single-node execution to 178 s for an 4-node execution (90% efficiency), and further to 89 s for an 8-node execution (still 90% efficiency). On the 450 MHz Pentium II cluster using SISCI/SCI, the efficiency is slightly lower (78% on 4 nodes), but the execution time is significantly better: the program runs in 273 s on 1 node, and 89 s on 4 nodes. Observe that the multithreaded program on one 450 MHz Pentium II node follows a more efficient search path for the particular problem being solved than its serial, single-threaded version reported in Section 4.1.

The efficiency decreases slightly as the number of nodes increases on a cluster. This is due to an increasing number of communications that are performed to the single node holding the current best answer. With a smaller number of nodes, there are more threads per node and a greater chance that, on a given node, requests by multiple threads to fetch the page holding the best answer can be satisfied by a single message exchange on the network. (That is, roughly concurrent page requests at one node may be satisfied by one message exchange.)

We believe these results indicate the strong promise of our approach. However, further study is warranted. We plan to investigate the performance under

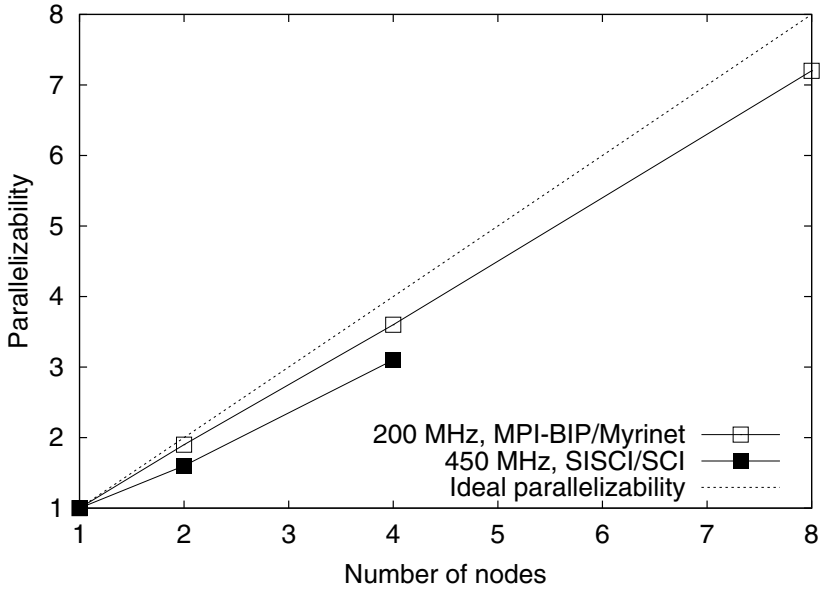

Fig. 3. Parallelizability results for the multithreaded version of our Java program solving the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Tests have been done on two cluster platforms: 200 MHz Pentium Pro using MPI-BIP/Myrinet and 450 MHz Pentium II using SISCI/SCI. The program is run in all cases with 64 threads.

Hyperion/PM2 of additional Java multithreaded programs, including applications converted from the SPLASH-2 benchmark suite.

5 Related Work

The use of Java for distributed parallel programming has been the object of a large number of research efforts during the past several years. Most of the recently published results highlight the importance of *transparency* with respect to the possibly distributed underlying architecture: multithreaded Java applications written using the shared-memory paradigm should run unmodified in distributed environments. Though this goal is put forward by almost all distributed Java projects, many of them fail to fully achieve it.

The JavaParty [13] platform provides a shared address space and hides the inter-node communication and network exceptions internally. Object and thread location is transparent and no explicit communication protocol needs to be designed nor implemented by the user. JavaParty extends Java with a preprocessor and a run-time system handling distributed parallel programming. The source code is transformed into regular Java code plus RMI hooks and the

latter are fed into Sun's RMI compiler. Multithreaded Java programs are turned into distributed JavaParty programs by identifying the classes and objects that need to be spread across the distributed environment. Unfortunately, this operation is not transparent for the programmer, who has to explicitly use the keyword `remote` as a class modifier. A very similar approach is taken by the Do! project [10], which obtains distribution by changing the framework classes used by the program and by transforming classes to transparently use the Java RMI, while keeping an unchanged API. Again, potentially remote objects are explicitly indicated using the `remote` annotation.

Another approach consists in implementing Java interpreters on top of a distributed shared memory [6, 17] system. Java/DSM [17] is such an example, relying on the Treadmarks distributed-shared-memory system. Nevertheless, using an off-the-shelf DSM may not lead to the best performance, for a number of reasons. First, to our knowledge, no available DSM provides specific support for Java consistency. Second, using a general-purpose release-consistent DSM sets up a limit to the potential specific optimizations that could be implemented to guarantee Java consistency. Locality and caching are handled by the DSM support, which is not flexible enough to allow the higher-level layer of the system to configure its behavior.

cJVM [2] is another interpreter-based JVM providing a single image of a traditional JVM while running on a cluster. Each cluster node has a cJVM process that implements the Java interpreter loop while executing part of the application's Java threads and containing part of the application objects. In contrast to Hyperion's object caching approach, cJVM executes methods on the node holding the master copy of the object, but includes optimizations for data caching and replication in some cases.

Our interest in computationally intensive programs that can exploit parallel hardware justifies three main original design decisions for Hyperion. First, we rely on a Java-to-C compiler to transform bytecode to native code and we expect the compilation cost will be recovered many times over in the course of executing such programs. We believe this approach will lead to much better execution times compared to the interpreter-based approaches mentioned above. Second, Hyperion uses the generic, multi-protocol DSM-PM2 run-time system, which is configured to specifically support Java consistency. Finally, we are able to take advantage of fast cluster networks, such as SCI and Myrinet, thanks to our portable and efficient communication library provided by the PM2 run-time system.

6 Conclusion

We propose utilizing a cluster to execute a single Java Virtual Machine. This allows us to run Java threads completely transparently in a distributed environment. Java threads are mapped to native threads available on the nodes and run with true concurrency. An original feature of our system is its use of a Java-to-C compiler (and hence of machine code). Hyperion's implementation supports a

globally shared address space via the DSM-PM2 run-time system that we configured to guarantee Java consistency. The generic support provided by DSM-PM2 allowed us to implement Java-specific optimizations that are not available in standard DSM systems, such as Treadmarks. Thanks to the portability of the PM2 run-time support, the full system we present is available on top of several UNIX systems and can use a large variety of network protocols. To evaluate our approach, we compare serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. We report good parallelizability on two platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

References

[1] Gabriel Antoniu, Luc Bougé, and Raymond Namyst. Generic distributed shared memory: the DSM-PM2 approach. Research Report RR2000-19, LIP, ENS Lyon, Lyon, France, May 2000.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.

[3] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. Held in conjunction with IPPS/SPDP 1999., Springer-Verlag.

[4] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 91–100, Palo Alto, California, February 1998.

[5] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10:1125–1242, 1998.

[6] X. Chen and V. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1998.

[7] A. Ferrari. JPVM: Network parallel computing in Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 245–249, Palo Alto, California, 1998.

[8] V. Getov, S. Flynn-Hummell, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 45–54, Palo Alto, California, February 1998.

[9] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addision-Wesley, Reading, Massachusetts, 1996.

[10] P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *High-Performance Computing and Networking (HPCN '98)*, volume 1401 of *Lect. Notes in Comp. Science*, pages 628–637. Springer-Verlag, 1998.

[11] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Third Usenix Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[12] R. Namyst and J.F. Mehaut. PM^2 : Parallel Multithreaded Machine: A computing environment for distributed architectures. In *ParCo'95 (Parallel Computing)*, pages 279–285. Elsevier Science Publishers, September 1995.

[13] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.

[14] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Third Usenix Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[15] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on Myrinet. In *Proceedings of First Workshop on Personal Computer Based Networks Of Workstations*, volume 1388 of *Lect. Notes in Comp. Science*. Springer-Verlag., April 1998.

[16] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *International Symposium on Computer Architectures*, pages 256–266, Gold Coast, Australia, May 1992.

[17] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of the Workshop on Java for High-Performance Scientific and Engineering Computing*, Las Vegas, Nevada, June 1997.