The Hyperion system: Compiling multithreaded Java bytecode for distributed execution

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

The Hyperion system: Compiling multithreaded Java bytecode for distributed execution ⋆

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

[a]*LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.*

[b]*Dept. Computer Science, Univ. New Hampshire, Durham, NH 03824, USA.*

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 parallelism, 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 communication interfaces 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 communication interfaces: SISCI/SCI and MPI-BIP/Myrinet.

Key words: Java, compiling, distributed shared memory, Java consistency, multithreading, Hyperion, PM2

1 Introduction

The Java programming language is an attractive vehicle for constructing parallel programs to execute on clusters of computers. The Java language design

⋆ A preliminary version of this work has been presented at the *Euro-Par 2000* Conference, Munich, Germany, August 2000.

[1] Contact: Gabriel.Antoniu@ens-lyon.fr.

reflects two emerging trends in parallel computing: the widespread acceptance of both a thread 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., [1–5]) or by grafting an existing message-passing library (e.g., [6,7]) 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 parallelism. The separate 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. (See [8] for a comparison of our work with an RMI-based approach.)

Our approach is most closely related to efforts to implement Java interpreters on top of a distributed shared memory [9–11]. However, we are interested in computationally intensive programs that can exploit parallel hardware. We expect that the cost of compiling to native code will be recovered many times over in the course of running 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 [13]. Another important advantage of PM2 is its high portability on several UNIX platforms and on a large variety of communication interfaces and protocols (BIP, SCI, VIA, MPI, 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 communication interfaces: the SISCI in-

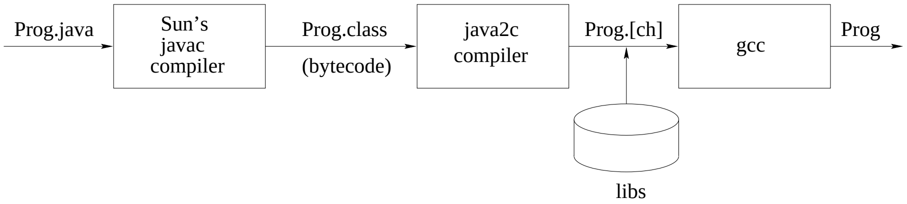

Fig. 1. Compiling Java programs with Hyperion

terface on top of a SCI network [14] and the MPI-BIP interface on top of a Myrinet network [15].

2 The Hyperion system

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.

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 [16] or Toba [17] 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.

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 other releases of Java 2 (e.g., Sun JDK 1.2) will be more amenable to being compiled by java2c.

Thread subsystem The thread module provides support for lightweight threads, on top of which Java threads can be implemented. This support includes thread creation/destruction and thread synchronization using mutexes. For portability reasons, we model the interface to this subsystem on the core functions provided by POSIX threads. Additionally, the thread subsystem provides an API for thread migration, which we plan to use in future investigations of dynamic and transparent application load balancing.

Communication subsystem The communication subsystem supports message transmission between the nodes of a cluster. The interface is based upon message handlers being asynchronously invoked on the receiving end. This 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.

loadIntoCache	Load an object into the cache
invalidateCache	Invalidate all entries in the cache
updateMainMemory	Update main 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
Interface between the Hyperion memory subsystem
and the implementation layer.

Memory subsystem The Memory subsystem is responsible for implementing the Java memory model. Its interface is based upon the operational specification of the Java memory model [18]. The model utilizes both a main memory and thread caches. Table 1 lists the key primitives of the Hyperion memory subsystem. These primitives conceptually manipulate the calling thread's cache and/or the main memory. (The Hyperion implementation of the Java memory model is discussed in detail in Section 4.)

Hyperion's memory subsystem 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.

Note that the whole design of the memory subsystem (and the corresponding API) is based on objects. However, the DSM support may be page-based. The memory subsystem interface hides such details from the rest of the system. More information about this point is given in Section 3.

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 investigation.

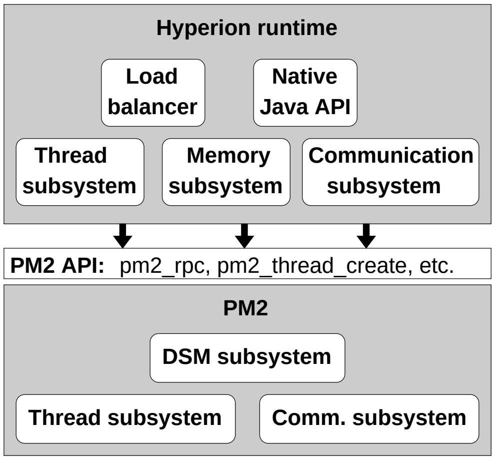

Fig. 2. Overview of the Hyperion software architecture

3 Implementing Hyperion on top of PM2

As specified in the previous section, each high-level component of the Hyperion runtime (namely the thread, communication and memory subsystems) requires that a number of functionalities be supported by the (lower) implementation layer. The current implementation of the Hyperion runtime is based upon the PM2 distributed multithreaded environment (Figure 2), which provides the features needed by all subsystems. Nevertheless, other implementations are possible (an alternative implementation of the memory subsystem is currently under study). In this paper, we focus on the PM2 implementation.

PM2's programming interface allows user-level threads to be created locally or remotely and to communicate through Remote Procedure Calls (RPCs). Besides, PM2 provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such a functionality is typically useful to implement dynamic load-balancing policies. Finally, the access to a common global address space is available via a *distributed shared memory* facility: the *DSM-PM2* [13] layer.

Most Hyperion run-time primitives in the thread, communication and shared memory subsystems are implemented through direct mapping onto the corresponding PM2 routines.

Thread subsystem The thread component of Hyperion is a very thin layer that interfaces to PM2's thread library, called *Marcel*. 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 to the thread management functions (e.g., thread creation).

Communication subsystem The communication component of Hyperion is implemented using PM2 remote procedure calls, 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 called *Madeleine* [19] that provides an efficient interface to a wide range of high-performance communication libraries, including low-level ones. The following communication interfaces are currently supported: BIP on top of a Myrinet network, SISCI on top of a SCI network, VIA, the Virtual Interface Architecture, MPI and TCP.

Memory subsystem The memory management primitives described in Table 1 are implemented on top of PM2's distributed-shared-memory layer, DSM-PM2 [13]. DSM-PM2 provides a portable, configurable implementation platform for multithreaded DSM consistency protocols. It has been designed to be generic enough to support multiple consistency models. Besides, alternative protocols are available for a given consistency model, thereby enabling the programmer to possibly tune the DSM system according to the specific application needs. Currently, two built-in alternative protocols provide Sequential consistency, and two other provide Release consistency. Also, new consistency models and protocols 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. The library routines (used to bring a copy of a page to a thread, 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.

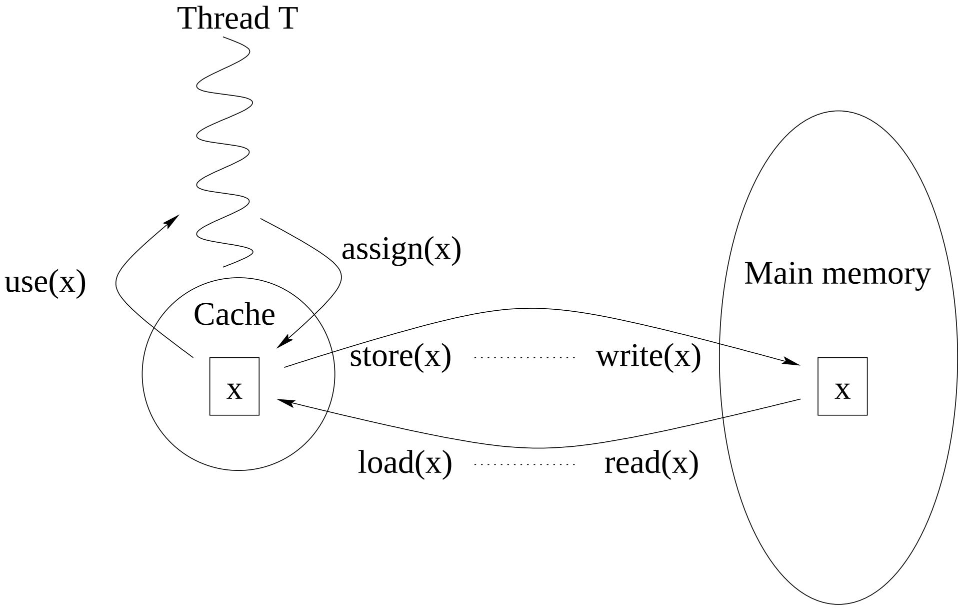

Fig. 3. Interactions between the thread cache and the main memory as specified by the Java Memory Model

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 memory subsystem primitives of Hyperion (loadIntoCache, updateMainMemory, invalidateCache, get and put) have been implemented using this latter approach, according to the specification of the Java Memory Model, as detailed in Section 4.

4 Implementing the Java memory model on a distributed cluster

4.1 The Java memory model

A central aspect of Hyperion's design is concerned with the implementation of the abstract memory model of Java within a physically distributed environment. The Hyperion system must provide the illusion of a uniformly accessible, shared object memory, which is independent of the physical object locations across the cluster.

The Java Memory Model (JMM) allows Java threads to use their private memory to cache values retrieved from main memory. Caching greatly improves performance if the application exhibits temporal locality, by accessing a cached object multiple times before the cache is invalidated. The specification of the Java language [18, Chapter 17] is very careful to describe how a thread may utilize its cache. It specifies that each Java thread is conceptually equipped with a *local cache*, interacting with a common *main memory* (Figure 3). A thread may perform *use* and *assign* actions to access values for variables that have been cached locally. The *use* action retrieves a variable's value from the cache for use by the thread's execution engine. The *assign* action overwrites a variable's value in the cache with a new value provided by the execution engine. The transfer of a variable's value from the main memory to the cache of a thread is completed by the unique pairing of a *read* action performed by main memory with a *load* action performed by the thread. The *read* action retrieves the variable's value from main memory and transmits it to the thread. The *load* action accepts the value from main memory and puts it in the thread's cache location for the variable. The reverse transfer, from a thread's cache back to main memory, is completed by the unique pairing of a *store* action performed by the thread with a *write* action performed by main memory. The *store* action transmits to main memory the value for a variable residing in the thread's cache. The *write* action accepts the value transmitted by the thread and deposits it in the main memory location for the variable.

The JMM also defines synchronization actions. Java monitors are described in terms of locks and each Java object is associated with a lock, which is also stored in the main memory. A lock may only be held by a single thread at a time. (A thread requesting a lock held by another thread will block until the lock becomes available and the requesting thread is able to obtain it.) Threads may request *lock* and *unlock* actions and the main memory is responsible for completing these actions. When a thread acquires a lock, it must flush all variables from its cache. Before a thread releases a lock, it must first transmit back to main memory all values it assigned since it acquired the lock. These two rules ensure that the conventional use of locks to protect a shared variable, will work correctly. After a thread grabs the lock, the first reference to the variable will cause the variable value to be read from the main memory and loaded into the thread's cache. The variable can then be updated and, when the thread performs the unlock, the updated value will be stored and written back to main memory, so as to be visible to other threads.

4.2 A distributed implementation in Hyperion

In Hyperion, the main memory region for an object is on its *home node*, the node on which the object was created. The home node is in charge of managing the *master copy* of the object. Initially, the objects are stored on their home nodes and can then be replicated if accessed on other nodes. On *using* an object, for instance as the right-hand side of an assignment, the local copy of the object in the node's cache is accessed; if no copy is present, then one is loaded from the main memory using the loadIntoCache primitive. On *assigning* an object, for instance through the left-hand side of an assignment, the local copy of the object in the node's cache is assigned; if no copy is present, then a fresh copy is loaded.

For a thread executing a *lock* action, Hyperion first performs the code necessary for the thread to acquire the lock, then updates main memory with all assigned values in the cache (via the updateMainMemory primitive), and finally invalidates the cache (via the invalidateCache primitive). For a thread executing the *unlock* action, the implementation first updates main memory with all assigned values in the cache (again via the updateMainMemory primitive) and then performs the code necessary for the thread to release the lock.

Note that the JMM only requires that the cache be invalidated when a lock is acquired, but that Hyperion also updates main memory. The update of main memory is not explicit in the JMM rules in this situation, but it is implied by the separate rule that requires a *store* action be subsequently performed for a variable prior to a *load* action for the variable, if an *assign* action has been previously performed for the variable. If the update of main memory were not done prior to the cache invalidation, then any assigned values in the cache would be lost when the cache was invalidated, in violation of this rule.

Hyperion uses specific access primitives to shared data: the get and put primitives. This allows us to detect and record all modifications to locally cached objects using a bitmap. 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. When updating main memory, Hyperion identifies all updated variables in the cache. Then it further identifies the home for all such variables and generates RPCs to all the homes to transmit the updated values. The calling thread must block until all such RPCs are explicitly acknowledged with an answering RPC, which indicates that the updated values were received and stored at the home node. Acknowledgments are necessary in order to prevent a thread from continuing on and releasing the lock "early." Such an early release would lead to an error, if another thread acquired the lock and cached variables that had been updated by the other thread, but for which the new values had not yet been written into main memory.

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. This pre-fetching is Java compliant because "early" *read* and *load* actions are explicitly allowed, as long as they are performed after the last *lock* action performed by the thread. Since each *lock* action is implemented with an invalidation of the local cache, any values pre-fetched before the last *lock* are discarded at the time of the *lock*. Therefore, any values that are “hits” in the cache are guaranteed to be values fetched since the last *lock* was executed. Note that caching at the page level, rather than at the object level, does not affect the updating of main memory. Since in the JMM a *lock* or a *unlock* action causes *all* cached values that have been modified to be sent to main memory, treating cached memory in units of pages is Java compliant.

In the original description of the Java memory model, each thread is equipped with its private, *thread-level* cache. The design of Hyperion recognizes that for many applications performance may be improved by running many threads in parallel, many more than the number of available nodes in the cluster. To support this view, Hyperion enables all the threads running on a single node to share a common, *node-level* cache. Sharing a common cache among all the threads running at a single node enhances prefetching of objects, thus providing a better locality of access. Also, the size of the caches is a function of the number of nodes, instead of the number of threads as in the original description. This is a major deviation from the original description of the JMM, which deserves a detailed discussion. We postpone this aspect to the next section for the sake of clarity.

Node-level concurrency provides the potential for concurrent execution of implementation actions such as page fetching, cache invalidation or the update of main memory. To limit this potential, Hyperion employs mutexes, as well as more complex locking mechanisms.

- First, each cached page is protected by a mutex to serialize the updating of the page’s modification bitmap. This mutex is also locked prior to transmitting the modifications back to the home node. The mutex is not locked by the get primitive, since there is no conflict between reading a field and either updating the encompassing page’s bitmap or sending the page’s modifications back to the home node.
- Second, a single, node-level mutex is employed to prevent concurrent cache invalidations and main-memory updates. This mutex is necessary to protect the internal data structures used by the cache.
- Finally, a more sophisticated locking mechanism is employed that prevents any thread from doing a cache invalidation until all threads are in a position to safely allow the invalidation to proceed. This mechanism is similar to solutions to the “readers-and-writers problem” in that multiple threads (the readers) may access cached values but only one thread (the writer) at a time can invalidate the cache. In addition, there can be no threads accessing the cache when it is invalidated. Threads release their “reader” lock whenever they reach a point in the program where an object might “move”. This includes *lock* and *unlock* actions when the cache is invalidated, as well as the new operator, when the garbage collector might be invoked. This mechanism

N_0	$Home_x$	N_1
loadIntoCache(x)		loadIntoCache(x)
request x		request x
	send x (value 7) to N_0	
	send x (7) to N_1	
T_0: use x (7)		T_3: use x (7)
T_1: assign $x = 17$		T_4: assign $x = 19$
T_2: use x (17)		T_5: use x (19)
updateMainMemory(x)		updateMainMemory(x)
transmit x (17)		transmit x (19)
	receive x (17) from N_0	
	receive x (19) from N_1	

Fig. 4. A typical situation for threads interacting on a variable x.

Main Memory
read x (value 7) for T_0
read x (7) for T_3
write x (17) for T_1
read x (17) for T_2
write x (19) for T_4
read x (19) for T_5

Fig. 5. The serial ordering for variable x.

prevents a cache invalidate when a page fetch is underway and thus avoids the need to possibly discard arriving pages and re-issue page requests. In addition, concurrent page requests can be safely and simply satisfied by a single page fetch from the home node.

4.3 Node-level concurrency and node-level caches

Does the *node-level cache* approach of Hyperion comply with the *thread-level cache* specification of JMM? And does special care need to be taken to support concurrent access to the local cache?

One aspect of sharing the local cache is that it extends pre-fetching to cross thread boundaries, in contrast with the operational description of the JMM. Actually, one thread can cause an object to be cached. The cached object may be subsequently accessed by another thread executing on the same node. The justification for this pre-fetching is the same as discussed in the preceding subsection. The key fact is that, if any thread invalidates the cache, the whole cache is invalidated for all threads. This ensures that any "hit" in the cache provides a valid value no matter which thread is executing the access. The values in the cache have all been loaded after the last cache invalidation performed by any thread.

A second aspect of cache sharing is that an *assign* performed by one thread on a given node will be "seen" earlier by other threads on the same node than by threads executing elsewhere in the cluster, again in contrast with the operational description of the JMM. Since the JMM utilizes a separate cache for each thread, it requires that, for a thread to *use* the value assigned by another thread, the value must be stored to main memory by the thread that performed the *assign* and then loaded from main memory by the thread that is performing the *use*. In Hyperion, however, the home location is not actually touched until there is a *lock* or a *unlock* executed on the node. This means that the JMM concept of main memory is not *directly* implemented in Hyperion by a set of physical *home locations*, one fixed location for each variable. In some circumstances, the value corresponding to the conceptual JMM main-memory copy of a given variable, may be read or written in the node-level cache, instead of its regular home location.

In fact, in Hyperion a variable may be cached on multiple nodes with multiple threads on each node concurrently accessing the local copy. The JMM requires that the main-memory actions performed on this variable be serializable. However, in Hyperion the actions are potentially performed concurrently by processors on different nodes. The key element to the argument that the Hyperion approach is JMM compliant is that the required serial ordering of JMM main-memory actions *for a given variable* can always be constructed from the node-level traces of the Hyperion implementation's manipulation of that variable at run time.

In general the serial ordering of the main-memory actions is constructed by splicing together the Hyperion traces for each node. The splicing is driven by the trace from the home node, using the calls to the loadIntoCache and updateMainMemory routines of Hyperion's runtime to control the merging from the other traces.

- A call to loadIntoCache for a variable x issues a read request from the local node to the home node for x; eventually, this home node serves the request by actually reading the value of x in from memory (*read* action of the JMM) and it sends it back to the requesting node; the value is eventually received and loaded into the cache (*load* action), where can subsequently be used (*use* action).

- A call to the updateMainMemory routine scans all the variables x which have been modified (*assign* action) at the local node since the last update; for each of them, it transmits the local value to the specific home node (*store* action); the home node eventually receives this value and actually updates its memory with it (*write* action).

To illustrate a typical situation, consider the sequence of actions displayed in Figure 4. Variable x is cached on two single-processor nodes, N_0 and N_1. There are three threads executing on each node: T_0, T_1 and T_2 on N_0; T_3, T_4 and T_5 on N_1. The home node for x is $Home_x$. Despite the concurrent assignments and the subsequent use of the assigned values by other threads, the required serial ordering of the main memory actions can still be constructed, as shown in Figure 5.

A full description of an algorithm for constructing a serial ordering of the JMM main-memory actions, given a set of Hyperion node-level traces, is provided in [20]. We are currently working on a formal description of this algorithm.

5 Performance evaluation: minimal-cost map-coloring

5.1 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 communication interface [21]. 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 communication interface. 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. 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. 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.

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 using the thread migration features of PM2.)

5.2 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 gcc 2.7.2.3 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. Even if the extra cost of a page fault is paid for every access to a non-cached remote object, this alternative technique avoids the high cost of the in-line checks. Such a comparison can be conveniently made using DSM-PM2’s generic support. An extensive study of the relative efficiency of *in-line* vs. *page-fault* checks, based on five other applications, can be found in [22]. There, the page-fault version usually performs significantly better than the in-line check version.

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

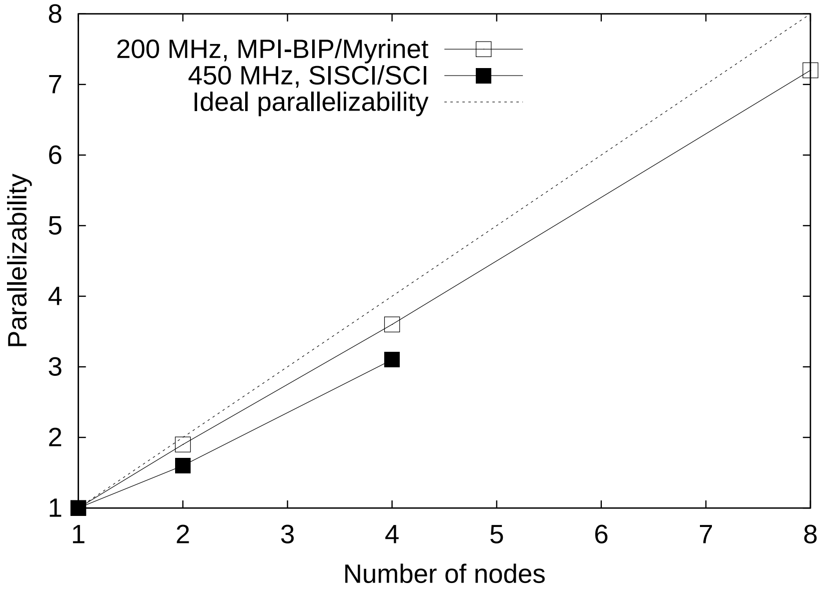

Fig. 6. Parallelizability results for the multithreaded version of our Java program solving the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Tests have been done on two cluster platforms: 200 MHz Pentium Pro using MPI-BIP/Myrinet and 450 MHz Pentium II using SISCI/SCI. The program is run in all cases with 64 threads.

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.

5.3 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 5.1. Parallelizability results are presented in Figure 6: 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 5.2. 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 Hyperion/PM2 of additional Java multithreaded programs, including applications converted from the SPLASH-2 benchmark suite.

6 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 [4] 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 [3], 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 [10,11] system. Java/DSM [11] 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 the Java memory model. 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 [9] 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 running 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.

7 Conclusion

The goal of the Hyperion project is to provide the software infrastructure to use Java to program low-cost, high-performance clusters of commodity processors. Hyperion supports viewing a cluster as executing a *single* Java virtual machine. We believe that transparently enabling the use of such clusters will make Java an attractive language for high-performance computing to a large class of programmers.

In Java, the notion of concurrency has been included from the very beginning in the language specification, both at the user level and within the bytecode. Concurrency is exposed to the user through *threads*, which share a common address space. The standard library provides a number of facilities to start a thread, suspend or kill it, switch control between threads, etc., and the *Java memory model* specifies how threads may interact with the common memory. It is thus possible to leverage this feature to directly map a multithreaded

Java program onto a cluster. Faster execution is obtained by mapping the original Java threads onto the native threads available in the cluster. These threads are spread across the processing nodes to provide actual concurrent execution and load balancing. The Java memory model is implemented by a distributed shared memory (DSM) substrate, so that the original semantics of the language is kept unchanged.

Portability has been a major objective in the design of Hyperion. We did not commit to any specific flavor of Unix, nor any processor architecture, nor any specific type of interconnection network and protocol. To achieve this challenge, the Hyperion platform has been built on top of a portable runtime environment called *DSM-PM2*, which is portable across a wide spectrum of high-performance networks, such as SCI, Myrinet, and Gigabit-Ethernet, and can be used with most common communication interfaces, such as TCP, MPI and VIA. While the system currently runs on clusters of Linux PCs, we expect to be able to use Hyperion on nearly any UNIX-based cluster or parallel computing environment without any significant modification.

Our benchmarking results demonstrate this portability as we ran an unmodified program on two clusters utilizing different processors, different network technologies and different communication protocols. Our results also identify two key needs to enable high-performance *serial* execution of Java code: the elimination of in-line checks for object locality and the optimization of array-bound checking. Finally, we presented parallelizability results that demonstrate good parallel efficiency for our benchmark application on the available clusters.

Hyperion therefore demonstrates that unmodified, standard Java programs can run effectively and transparently on clusters. Java threads can be implemented with high-performance, low-level threads that can be mapped to different processors for true parallelism and the Java memory model can be efficiently implemented in a distributed-memory environment. This combination of performance, transparency and portability makes Java an attractive alternative for programming clusters of computers.

Acknowledgments

We thank the anonymous referees for their remarks, which helped us improve the quality of this paper.

The Hyperion-PM2 collaboration was supported by funding from the NSF and INRIA via the USA-France Cooperative Research program.

References

[1] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, D. Gannon, Java RMI performance and object model interoperability: experiments with Java/HPC++, Concurrency: Practice and Experience 10 (11-13) (1998) 941–955.

[2] D. Caromel, W. Klauser, J. Vayssière, Towards seamless computing and metacomputing in Java, Concurrency: Practice and Experience 10 (11-13) (1998) 1043–1062.

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

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

[5] J. Maassen, T. Kielmann, H. Bal, Efficient replicated method invocation in Java, in: Proc. of the ACM 2000 Java Grande Conference, San Francisco, California, 2000, pp. 88–96.

[6] A. Ferrari, JPVM: Network parallel computing in Java, Concurrency: Practice and Experience 10 (11-13) (1998) 985–992.

[7] V. Getov, S. F. Hummel, S. Mintchev, High-performance parallel programming in Java: exploiting native libraries, Concurrency: Practice and Experience 10 (11-13) (1998) 863–872.

[8] T. Kielmann, P. Hatcher, L. Bougé, H. Bal, Enabling Java for high-performance computing: Exploiting Distributed Shared Memory and Remote Method Invocation, Communications of the ACM (2001). Special Issue on Java for High Performance Computing, to appear.

[9] Y. Aridor, M. Factor, A. Teperman, cJVM: A single system image of a JVM on a cluster, in: Proc. of the International Conference on Parallel Processing, Fukushima, Japan, 1999, pp. 4–11.

[10] X. Chen, V. Allan, MultiJav: A distributed shared memory system based on multiple Java virtual machines, in: Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, 1998, pp. 91–98.

[11] W. Yu, A. Cox, Java/DSM: A platform for heterogeneous computing, Concurrency: Practice and Experience 9 (11) (1997) 1213–1224.

[12] R. Namyst, J.-F. Méhaut, PM2: Parallel multithreaded machine, a computing environment for distributed architectures, in: Parallel Computing (ParCo '95), Elsevier Science Publishers, 1995, pp. 279–285.

[13] G. Antoniu, L. Bougé, DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols, in: Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01), San Francisco, 2001, to appear.

[14] H. Hellwagner, A. Reinefeld (Eds.), SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters, Vol. 1734 of Lect. Notes in Comp. Science, Springer-Verlag, 1999.

[15] L. Prylli, B. Tourancheau, R. Westrelin, The design for a high performance MPI implementation on the Myrinet network, in: Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99), Vol. 1697 of Lect. Notes in Comp. Science, Springer Verlag, Barcelona, Spain, 1999, pp. 223–230.

[16] G. Muller, U. Pagh Schultz, Harissa: A hybrid approach to Java execution, IEEE Software 16 (2) (1999) 44–51.

[17] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, S. Watterson, Toba: Java for applications – a way ahead of time (WAT) compiler, in: Proc. of the Third Conference on Object-Oriented Technologies and Systems, USENIX Association, Portland, Oregon, 1997, pp. 41–53.

[18] J. Gosling, W. Joy, G. Steele Jr., The Java Language Specification, Addison-Wesley, Reading, Massachusetts, 1996.

[19] L. Bougé, J.-F. Méhaut, R. Namyst, Efficient communications in multithreaded runtime systems, in: Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99), Vol. 1586 of Lect. Notes in Comp. Science, Springer-Verlag, San Juan, Puerto Rico, 1999, pp. 468–482.

[20] P. Hatcher, On the correctness of a cluster implementation of Java, Technical Report TR 00-05, University of New Hampshire (Jul. 2000).

[21] L. Prylli, B. Tourancheau, BIP: A new protocol designed for high performance networking on Myrinet, in: Proc. of First Workshop on Personal Computer Based Networks Of Workstations (PC-NOW '98), Vol. 1388 of Lect. Notes in Comp. Science, Springer-Verlag, 1998, pp. 472–485.

[22] G. Antoniu, P. Hatcher, Remote object detection in cluster-based Java, in: Proc. 3rd Int. Workshop on Java for Parallel and Distributed Computing (JavaPDC '01), San Francisco, 2001, to appear.