TECHNICAL REPORT 11/2008

Accelerating the parallel distributed execution of Java HPC applications

BSC-UPC

COMPUTER SCIENCES

Enric Tejedor
Rosa M. Badia
Pere Albert

Accelerating the parallel distributed execution of Java HPC applications

Enric Tejedor, Rosa M. Badia (BSC)
Pere Albert (IBM - i3@BSC)

November 20, 2008

Abstract

In order to speed up the execution of Java applications, JIT compilers compile method bytecodes into native code at run time. Once a method is compiled, subsequent executions of this method will run a generated machine-dependent code, which is generally faster than interpreting bytecodes. In addition, JIT compilers can apply several optimizations to the code to increase the performance even further.

When parallelizing the execution of a Java application over a distributed infrastructure such a Grid, a JVM is created on each resource node to run a part of the application. The JIT compiler of each of these JVMs works independently from the others, thus paying the compilation cost and the time to reach the peak performance in every machine. Besides, in some cases, the execution nodes are not dedicated or not directly accessible; this prevents from maintaining a persistent JVM in each node during all the application execution, and gives little opportunity for the JIT compiler to perform mid-to-long term code optimizations.

This document proposes a couple of techniques to fully benefit from JIT optimizations when distributing the execution of Java applications. These techniques could be integrated in COMP Superscalar, a framework that allows to run user-selected parts of a Java application in a set of distributed parallel resources.

1 Summary

This document describes the challenges of parallelizing the execution of Java High-Performance-Computing (HPC) applications over an infrastructure like the Grid, which can be defined as a set of geographically-distributed resources that can collaborate to execute an application. In our group, we have developed a framework called COMP Superscalar [2] (COMPSs), which allows the user to select computationally-intensive methods of a Java application to be run remotely on the Grid instead of locally.

At execution time, COMPSs transforms the local invocations to the selected methods into the creation of tasks that will be run remotely. Such tasks are scheduled in the available Grid resources, called workers. Every time that a task (i.e. an invocation to a selected method) must be run on a worker resource, a new Java Virtual Machine is launched on that worker, then the method is executed and finally the JVM is destroyed. This implies that the worker JVMs are transient. The motivation to use this execution model with transient JVMs is the characteristics of a Grid: in some cases, Grid resources can't be dedicated to hold a persistent JVM during all the application execution, or they are not directly accessible, because they are hidden behind a front-end node that receives the execution requests and forwards them to a local scheduler, which decides the worker node where the job runs.

Such execution model is well adapted to the Grid environment, but also it imposes a significant drawback for Java application: it limits the activity of the Just-In-Time compiler. Over the last decade, Java runtime vendors have developed dynamic compilers, known as Just-In-Time (JIT) compilers, e.g. the IBM Testarossa included in the IBM J9 JVM 1.5+ [4] or the Sun HotSpot JIT compiler [5]. Their purpose is to improve the performance of Java applications by compiling bytecodes to native code at run time, eventually optimizing this code later, while maintaning the portability that Java requires. The methods selected to be compiled are normally those with the highest frequency of invocation.

In the transient JVMs execution scenario, the JIT compiler has little opportunity to perform mid-to-long term optimizations to the task code. In order to benefit from such optimizations in COMPSs, the JVMs should rather be persistent: the first time that a method would be run on a host, the JVM would be created and then reused for subsequent method executions, which would allow the JIT to keep the jitted code of a task for later executions of the same task, and also to improve the code further.

As a response to that problem, we propose a couple of solutions. The first one involves the utilization of selective compilation techniques. When running an application with COMPSs, we know in advance which are the hot spots of the application: the computationally-intensive methods that the user has selected for execution on the Grid; thus, telling the compiler to compile (and possibly to optimize) this set of methods could reduce the execution time of the remote tasks. Moreover, in combination with this first technique, we would also use caches of compiled methods that could be shared by all the worker JVMs. The IBM J9 VM already provides the possibility to generate caches with JIT compiled code, the AOT caches, which can be stored as files. COMPSs would be in charge of transferring the code cache generated by one worker to another one, whose JVM would relocate and reuse that code to run its assigned task, thus avoiding possibly slow interpreted executions. Nevertheless, the level of optimization that

can currently be reached in an AOT method is low. In order to get better performance, the optimization level of the AOT code should be increased. For this reason, we are considering the possibility of creating a cache that stores highly-optimized JITted code.

The success of this work will depend on the capability of a JIT compiler to improve the performance of Java HPC applications with coarse-grain tasks as hot spots. Moreover, we have to evaluate the performance of our solution using selective compilation combined with AOT caches, as well as the feasibility of increasing the optimization level on this caches. If necessary, we could implement a prototype of highly-optimized code caches in the IBM J9 JVM, or alternatively in one of the open source research JVMs available, like the Jikes RVM [7] and its Quicksilver compiler [9].

2 Execution environment: the Grid and COMP Superscalar

This section presents the enviroment that we target to distribute and parallelize the execution of Java applications: Grid infrastructures managed by the COMP Superscalar framework.

2.1 The Grid

Grid computing [1] is a form of distributed computing that makes use of heterogeneous, loosely-coupled and geographically-distributed resources, such as computational servers and storage systems, which are connected to a network (public, private or the Internet).

From a user's point of view, a Grid is accessed as a single virtual supercomputer, formed by a set of physical resources acting in concert to run large applications composed by computationally-intensive tasks.

2.2 COMP Superscalar

COMP Superscalar [2] (COMPSs) is a framework that orchestrates the execution of Java applications on the Grid. The two main distinctive features of COMPSs are the programming model that it offers and its runtime.

- *Programming model*: let's assume that the user has a sequential Java application which invokes one or more methods that are potentially expensive in terms of computation. The programming model of COMPSs allows the user to select these methods to run them on a set of Grid resources, instead of locally. The selection is done by providing a Java interface that declares the methods, along with some simple metadata in the form of Java annotations. Concerning the application, *the user does not have to modify its original sequential code at all*, COMPSs

will be in charge of instrumenting it and running the invocations to the selected methods (what we call *tasks*) on the distributed resources.

- *Runtime*: COMPSs replaces the invocations to the selected methods by invocations to its runtime. When the application is running, for each call to a selected method, a task is created. The runtime of COMPSs discovers the dependencies between tasks, and with that information it builds a task dependency graph. The tasks of this graph are scheduled for execution on the Grid resources, trying to exploit the parallelism exhibited by the graph as much as possible. The runtime also manages the transfer of the input files of a task to the destination resource, the remote execution of the task and the collection of results at the end of the application.

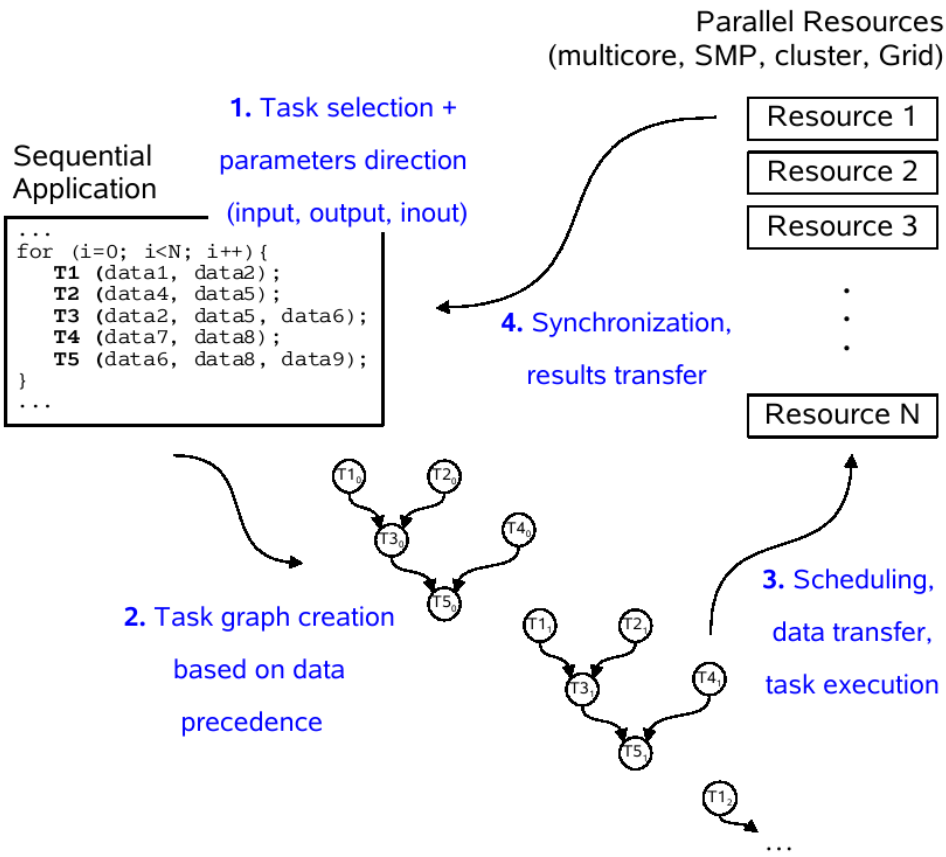

Figure 1: Programming model and runtime operation of COMP Superscalar

The current task execution model of COMPSs is based on a *transient workers* paradigm: whenever the runtime decides to run a given task in a remote Grid resource (what is called a *worker*), a new Java Virtual Machine

is launched on that worker, then the corresponding method is executed and finally the JVM is destroyed. This happens for each task, which implies that no persistent JVM is kept in the workers between task executions.

3 Problem: the influence of the JIT compiler

When a sequential Java application is run on a single resource with a unique JVM, the JIT compiler performs an analysis to detect the hot spots, selects methods for compilation and applies optimizations to the code if convenient.

The fact of distributing the execution of a Java application over N different resources implies having N independent JVMs that run a part of the application. Each of the JITs, therefore, perform a number of compilations/optimizations in its part of the code, possibly repeating work that other JITs have already done. Even worse, if no persistent JVM can be kept in the resources, JVMs are created and destroyed all the time during the execution of the application, thus limiting the operation of the JIT compilers.

The problem described above arises when COMPSs distributes the tasks over the Grid resources. Since we are in a transient JVM scenario, the executed methods will never be optimised at the level that they could be. Even if a JIT compiler produces a jitted version of a task method, when the method finishes the JVM is terminated, and for that reason the next task which is run on the same node will not have this jitted code available. The worst case happens when the JIT does not compile the task method before its execution and consequently the bytecode of the method is interpreted; this interpreted execution can be significantly slower than the one of a jitted method, and it will repeat for every execution of the task.

Note the contrast between the distributed execution and a sequential execution of the application in a single-node JVM. In the latter case, the jitted code produced when running a method can be used in subsequent invocations of the same method, and eventually an optimized jitted code can be generated.

As an example of the problem, we present here some test results of parallelizing an application with COMPSs. The application is called *Matmul* and multiplies two matrices. The matrices are divided into blocks, which are themselves smaller matrices of doubles. The tasks generated by Matmul work with blocks stored in files. Figure 2 depicts the speedup of Matmul depending on the number of worker processors, and considering two different JVMs: the IBM J9 VM 1.6 and the Sun HotSpot VM 1.6 in server mode. In both cases, the baseline is a sequential execution of Matmul in one node.

As can be seen in Figure 2, the results for the HotSpot VM in server mode show a poor speedup. The reason is that, when running the baseline with the Server HotSpot, the method that multiplies two matrix blocks is

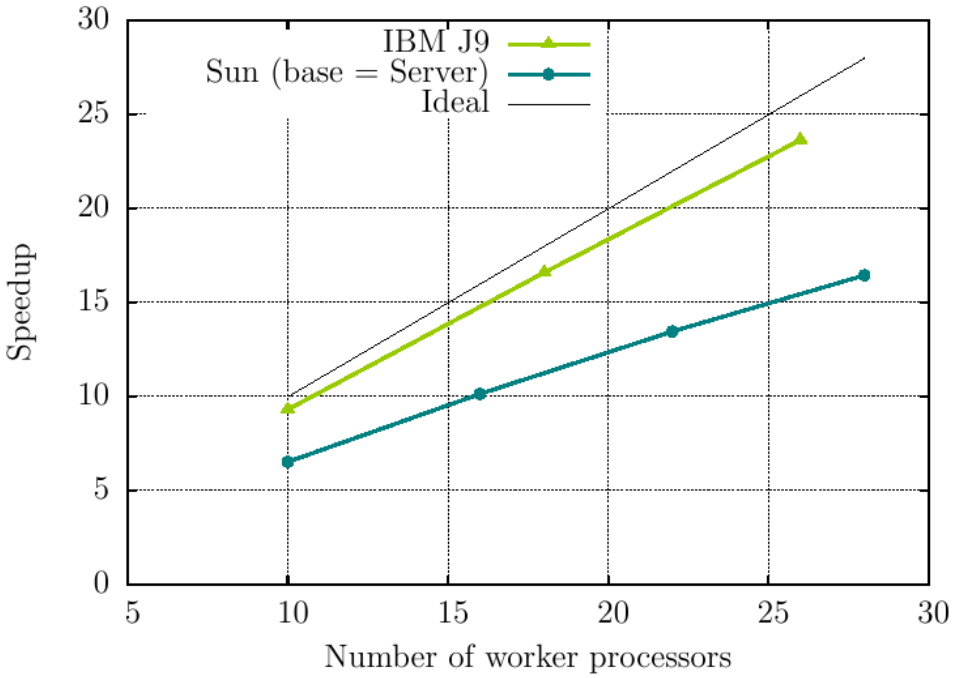

Figure 2: Speedup of the Matmul application for two different VMs

progressively optimized along several runs, until it reaches a notably better performance than that of the first run of the method. However, the distributed execution of Matmul cannot benefit from such progressive optimizations, because the transient JVMs only run the method once before they terminate.

On the contrary, the speedup obtained with the IBM VM is close to the ideal. The reason is that, when running the baseline with this VM, the execution time of the block multiplication method was more or less constant from its first run to the last, i.e. the performance obtained with the first jitted version of the method was not remarkably improved when applying higher levels of optimizations. Therefore, in the distributed execution, all the tasks had the same performance which was already much the best that the JIT compiler could get.

This example illustrates how the JIT compiler optimizations can influence the execution time of an application, and how missing these optimizations when distributing the application can lead to a noticeable underperformance. In conclusion, if we were capable of benefiting from aggressive optimizations also in the COMPSs worker JVMs, we would improve the execution time of applications whose tasks can be effectively optimized. In this sense, Section 4 proposes some solutions to address the issue.

4 Solutions

When running an application with COMPSs, we know in advance the methods whose performance is important: those that the user selects for execution on the Grid (the tasks). This means that the selected methods or any method that is called by them, directly or indirectly, are likely to be computationally intensive. Thus, the fact of optimizing them at a high level could represent a huge improvement in the overall performance of the application.

The next subsections describe two complementary techniques that could contribute to run the tasks code faster, by taking more advantage of compiled/optimized code.

4.1 Selective compilation

The IBM J9 JVM 1.5+ provides several undocumented command-line options to influence the compilation behaviour. One of them allows to specify a set of methods to be compiled. Optionally, two sub-arguments can be defined:

- *count*: the number of times that the method will run before it is compiled. If count=0, the application thread is halted before the execution of the method and then the method is jitted; after that, the application thread restarts and runs the jitted method. If count=n and n>0, the method is jitted asychronously with the application thread, so that the application thread may actually interpret the method a number of invocations before the jitted method is used (in fact, short running applications may end before the compilation completes).

- *optlevel*: the optimization level at which the method is compiled. The Testarossa JIT compiler has six levels: noOpt, cold, warm, hot, veryHot and scorching.

The syntax of this selective compilation option is:

$$java - Xjit :' \{a/b/C.methodName*\}(count = 0, optlevel = hot)'$$

In this example, we instruct the JIT compiler to compile any method whose name begins with 'methodName', belonging to the class 'a.b.C', at first touch ('count=0') and at optimization level 'hot'.

With selective compilation *we can ensure that a given method will never run in interpreted mode*, thus avoiding probably slow executions. Besides, we can try to improve the performance of the method even further with the 'optlevel' sub-argument. However, it is worth noting that good optimizations could require the method to run at least once, for the JIT compiler to collect run time information about the execution. At the first invocation of a

method, most fields and invokes referenced from within that method appear unresolved for the JIT, and this inhibits many JIT optimizations, such as inlining. Therefore, specifying 'count=0' along with a high optimization level as sub-arguments is unlikely to improve the performance very much.

Another important thing to mention is that, in COMPSs, it might not be helpful to compile the methods which are selected by the user to be run on the Grid. The reason is that such methods might not be themselves the hot spot of the application, but a method which is called from them. Since instructing the JIT to compile/optimize a method does not imply that the methods invoked from it will be also compiled/optimized, the gain in performance could be poor if the real computation is done inside an invoked method. Consequently, the task methods should be profiled first to find out where the hot spot resides, and then perform selective compilation accordingly.

At the light of what is explained above, we can conclude that selective compilation alone may not be completely effective. In a scenario with transient JVMs, even if we instruct the JIT to compile the task methods we would never have fully optimized executions, because the JIT would never have information about a previous execution to further optimize the code. A possible solution to mitigate that problem could be to have partially-persistent JVMs: tasks could be grouped in clusters and sent together for execution on the same JVM. Still, in this case all the JVMs would have at least one non-optimized execution of each task method. In this sense, the next subsection describes a complementary technique to fully benefit of JIT optimizations in the worker nodes.

4.2 Caches of methods

Another feature of the IBM J9 JVM is the Ahead-Of-Time (AOT) compilation [3]. If the AOT compiler is enabled, a data cache containing classes and compiled methods is generated and stored, either in memory or disk, while the application runs. Subsequent JVMs can share this cache and load AOT code from it without incurring the compilation overhead again.

AOT caches could be used in our case to hold the compiled code of the task methods, but they are not completely adequate for two reasons. First, the AOT compiler heuristically selects the methods to be compiled with the primary goal of improving startup time, whereas our goal is to achieve an optimized performance of the computationally intensive task methods. Second, and more important, *AOT code does not perform as well as highly-optimized JIT code.*

The reason of the AOT underperformance is that AOT code must persist across different program executions, and consequently it is not the result of aggressive and execution-dependent optimizations. However, and here resides the difference, *the task methods that COMPSs executes remotely are*

part of the same original sequential application, even if they are run remotely and separately, and therefore *they are likely to share the same execution parameters*. For instance, when running the matrix multiply application with COMPSs, where each task multiplies two blocks of the input matrices, the JIT may generate an optimized code for a task especially tuned for the concrete block size that is being used; since the block size is the same for all the tasks, this jitted code could be reused by other JVMs, deployed in other nodes, that run the same multiply task for another pair of blocks. On the contrary, the AOT compiler probably would not generate a code dependent on the block size, because AOT code is intended to be valid for different executions with different parameters.

Taking into account all these considerations, the best solution in our case would be to have *caches of highly-optimized JITted methods*. The idea is that *the code jitted by a worker JVM could be reused by another worker JVM running in the same architecture*, which would not pay the compilation cost nor possibly the time to reach the peak performance if the shared code is already fully optimized. In this execution model, the JIT compilers of the different JVMs would not be independent but collaborative, and the optimizations performed inside a JVM could spread to other JVMs in the system.

Concerning the possible scenarios, optimized code caches could make a great performance impact when having transient JVMs in the worker nodes. Instead of being limited by plain compilations with no execution information in all the nodes, now we could also have fully optimized task executions: from the moment that a JVM finishes a task and generates a jitted code, the rest of the tasks could take that code instead of starting from scratch all over again. Although this technique would have less influence in the persistent workers scenario, the cache sharing could improve the overall performance as well since the nodes could exchange more and more optimized versions of the task methods.

COMPSs could be in charge of mananing the caches. They could be stored in files and COMPSs would order their transfers between different nodes: COMPSs would be aware of where a given task code has been generated and it would send the cache to another node where the same task will run. In fact, caches could be themselves another scheduling parameter: besides considering data locality, COMPSs could also take into account cache locality, and schedule tasks in nodes where there is already a cache that contains the task code. Regarding the lifetime of these caches, they could last until the application ends, or also they could persist between different executions (in the case of code caches containing execution-dependent optimizations, though, the parameters should be the the same or similar in all the executions).

5 Key factors for the success of this work

There exist three main factors that will determine the success of this document's proposals. They are described in the next subsections.

5.1 Well-performing JIT compiler

First, the gain in performance of our solutions will be proportional to the efficiency of the JIT compiler. It is of great importance to have a JIT compiler which is capable of applying effective optimizations that reduce the execution time of the application hot spots. Only with such a JIT compiler, the selective compilation and the method caches will contribute to accelerate the remote execution of tasks.

5.2 Java HPC applications

Second, the performance of the JIT compiler is particularly important in the type of Java applications that we target: HPC applications composed of calls to one or more coarse-grain methods with a reasonable degree of parallelism between them. In fact, the applicability of our solutions depends on the performance that the JIT compiler exhibits on this kind of applications.

Therefore, part of the work to do consists in finding one or more Java HPC applications that can be run with COMP Superscalar and whose hot spots (the task methods) can be optimized by the JIT compiler. If such optimizations represent a considerable reduction of the execution time in a sequential run of the application, the gain in performance obtained when applying the proposed solutions in a distributed run will be significant.

5.3 Feasibility of increasing the optimization level in code caches

Last, whereas the options for selective compilation already exist in the IBM J9 VM (though not documented), at the best of our knowledge no vendor currently implements a cache of highly-optimized jitted code.

As explained in Section 4.2, the AOT caches of the IBM J9 VM are one of the options that we are considering, however the level of optimization that they offer is low. Similarly, the Sun HotSpot VM 1.5+ provides a feature called Class Data Sharing (CDS) [6], which allows to create an internal representation of a set of classes and dump it into a file, called 'shared archive'. The objective of this archive, like the AOT caches, is to reduce the startup time of Java applications.

Since none of the existing JVMs provides caches of highly-optimized jitted methods, one possibility could be to implement a prototype of such caches. For that purpose, we could try to increase the optimization level of the AOT caches generated by the IBM J9 JVM. Alternatively, we could

extend a research virtual machine which comes with a JIT compiler, like the Jikes RVM [7] or Kaffe [8], which are open source VMs for research and education. In both cases, we would have to evaluate the effort that the implementation of such a cache requires and the feasibility of this solution, especially the difficulties arisen when trying to relocate a highly-optimized jitted code generated by a certain VM into another VM. In this sense, a good reference could be the work on the Quicksilver compiler [9], which was used together with the Jikes RVM to generate quasi-static images of optimized code.

6 Conclusions

This document has presented a major challenge for the execution of Java applications in a distributed parallel infrastructure like the Grid: exploiting the run time optimizations that the JIT compiler performs to the code. We have seen how the heterogeneity of the Grid sometimes prevents from having dedicated JVMs in the execution nodes, and how this can limit the activity of the JIT compiler.

As proposed solutions, we have described the selective compilation options provided by the IBM J9 VM and the creation of method caches. These proposals intend to steer the compilation process of all the JIT in the distributed system and make them collaborate.

Some key factors have to be evaluated before the beginning of this work, mainly the ability of JIT compilers to improve the performance of Java HPC applications and the feasibility of applying the proposed solutions.

References

[1] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1998, ISBN 1-55860-475-8.

[2] E. Tejedor and R. Badia, *COMP Superscalar: Bringing GRID superscalar and GCM Together*, in 8th IEEE International Symposium on Cluster Computing and the Grid, May 2008.

[3] *IBM Diagnostics Guide 6.* http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html

[4] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley, *Experiences with multi-threading and dynamic class loading in a Java Just-In-Time compiler*, in CGO 06: Proceedings of the International Symposium on Code Generation and Optimization. Washington, DC, USA: IEEE Computer Society, 2006, pp.8797.

[5] *The Java HotSpot Performance Engine Architecture*, White Paper. http://java.sun.com/products/hotspot/whitepaper.html

[6] *Class Data Sharing in the Sun HotSpot VM.* http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html

[7] *Jikes Research Virtual Machine.* http://jikesrvm.org

[8] *Kaffe VM.* http://www.kaffe.org/

[9] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta, *Quasi-Static Compilation for Java*, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA00), Minneapolis, October 2000, pp. 6682.