Multiprocessor support in Java

Francisco J. Novo-Sánchez, Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza
Software Engineering Department
University of Granada
Granada, Spain
fnovo@ugr.es, sandra@ugr.es, jholgado@ugr.es

Abstract—In this study we analyze different alternatives to develop applications in multiprocessor systems using Java. We include a study about the Real Time Specification for Java (RTSJ) and improvements included in the version 1.1 to optimize the resources available in this kind of systems. We analyze how JamaicaVM, a commercial implementation of the RTSJ, uses these improvements and what are their limitations. We have carried out an empirical evaluation using a case study. This evaluation uses the execution times obtained with the parallel execution of the case study with respect the sequential ones. With the analysis of results, it is demonstrated the improvement in the performance obtained with the parallelization of the problem.

Keywords—Java; multiprocessor; real time; RTSJ; Jamaica; OpenMPI

I. INTRODUCTION

There is a large increase of systems based on multiprocessor architectures, both desktop and embedded systems. This increasing has resulted in the adaptation of platforms, tools and programming languages to include new extensions, primitive methods and models for software development in this kind of systems. These changes are necessary to obtain an optimal performance in the use of all the resources of such systems.

There are programming languages with direct access to hardware resources, e.g., C or C++. These languages have basic primitive methods that allow developers to use the resources of different types of processors. However, other high level languages, such as Java, have to adapt their APIs to support this type of architectures and to maximize the performance that developers can obtain in their applications.

Different specifications of Java platforms have to include support elements to optimize the application performance on multiprocessor systems. For example, the Real-Time Specification for Java (RTSJ) in its version 1.1 includes new models to set the affinity of threads into the multiple processors available [1]. But the efforts of the Java developer community are beyond the language specification. The community has also developed specific multiprocessor chip for real-time Java [2], designed simulation tools to enable analysis and evaluation of Java algorithms on multiprocessor systems [3], and designed new methodologies to promote compliance with restrictions of time and memory usage in multiprocessor systems [4]. OpenCL, Open Compute Language [5], is worth mentioning regarding open proposals for parallel programming on heterogeneous devices. OpenCL was initially developed by Apple Inc. but currently is managed by the non-profit consortium Khronos Group. The use of OpenCL is widespread in programming GPUs [6]. Even Intel, which has its own development environment and its own API to program graphics chips called CUDA, supports OpenCL.

The goal of this study is to analyze the alternatives provided by Java to develop applications with multiprocessor or multicore support. Due to the high relevance of real time aspects in distributed computing, we have performed a comprehensive analysis of the real time specification and the facilities included on it to develop applications in such systems. Besides JavaSE and Java interface of OpenMPI, we have also analyzed the mechanisms of parallelization in JamaicaVM, the RTSJ commercial implementation of Aicas. Related to the conceptual and theoretical analysis of these concepts, we have also conducted an empirical evaluation using different implementations of a basic problem that can be optimized using parallelism. We compare the parallel execution time of the program with the time obtained using a sequential version. It is demonstrated the improved performance provided by the parallel version in multiprocessor systems.

The paper is organized as follows: Section II presents the Real-Time Specification for Java (RTSJ) and its main features; in Section III we conduct a comparison between different options to implement parallelism with multiprocessor support in Java; the assessment undertaken and the results of tests carried out are shown in Section IV; and finally, in Section V, we present main conclusions.

II. REAL-TIME SPECIFICATION FOR JAVA

Java has grown since its origin and it has adapted to different development scenarios, such as desktop, web, mobile, and critical applications [6]. Java has developed a huge variety of specifications and implementations taking into account the specific characteristic of these scenarios. For example, there is a Java specification to develop applications for intelligent cards (JavaCard) [7] and another one for embedded systems (J2ME) [8]. Currently, the Java community is doing great efforts to adapt the specification to improve the development in real-time embedded systems. To cover the necessities in this kind of systems, a group of experts in real-time and Java (RTJEG) developed a specification to extend the language with the support of the Java Community Process (JCP) [9]. This specification has new mechanisms and tools specifically designed to develop real-time applications. It is the Real-Time Specification for Java (RTSJ) [10]. It is available the version 1.1 and it is responsibility of the group JCP-282 [11]. The RTSJ follows next design principles [12]:

...
a) Predictable execution.
b) Applicable in any Java execution environment.
c) Retrocompatibility with other Java versions.
d) Not include syntactic extension to the Java language.
e) Fill all the necessities of the real-time systems allowing adding new advances charasteristic in future implementations.
f) Permit variations related with different implementation decisions.
g) WORA (Write Once, Run Anywhere) without impact in the performance.

B. General summary of the affected areas

There are several areas where Java does not give support for the real-time. For example, memory management (garbage collection or GC), scheduling, synchronization of threads, asynchronous events management, and accessing to the physical memory [13]. In the following subsections, we analyze the elements of the RTSJ related with the areas above mentioned.

1) Thread scheduling

There are several scheduling models with significant differences between them. Each model has its own industrial real-time application. The specification gives an underlined scheduling mechanism that can be mapped with the thread management of the operative system. However, the specification does not specify how all the scheduling mechanisms work. The RTSJ has been designed to provide unanticipated scheduling algorithms. The implementation permits to assign proper parameters depending on the specific mechanism used. Moreover, it also gives methods to start, manage, follow and kill the real-time threads. Additionally, the specification is ready to plug in a new mechanism adding additional modules with different scheduling plans at a later time.

Due to the actual practice of developing real-time applications, the RTSJ needs a default base scheduler in all its implementations. The base scheduler is well known by the real-time developers. It is priority-based, with at least 28 unique priorities.

2) Memory management

The automatic management of the memory is an important characteristic of Java. Accordingly, the RTSJ may to allow implementing memory management by the underline system, releasing the developer of this task. There are several algorithms to carry out the memory management, known as garbage collection (GC).

The GC is applied in different ways depending on the implementation style and the specific type of real-time system to develop. It is specified a system to assign the memory independently of the GC algorithm. This allows defining the behavior of the collector during the execution time, preemption, and dispatching of real-time threads. Also, it permits to delimit the memory area of the new objects without the collector interference.

3) Synchronization and resource sharing

Real-time systems have high complexity with respect the synchronization due to the priority inversion. The less intrusive specification with a safety real-time synchronization imposes some requirements to the implementation of the synchronized directive. It may to include at least a mechanism to avoid priority inversion between the real-time threads that share a resource, such as priority inheritance or priority roof. However, sometimes it is not enough using the synchronized directive with an algorithm to avoid priority inversion. To solve this situation, the specification allows to give more priority to threads than to the GC, and to use a set of wait-free queues. This ensures that the GC does not delay threads execution.

4) Asynchronous events handling

As usual, real-time systems interact with the environment. The environment is asynchronous with respect to the execution of the thread code. The RTSJ generalizes the Java mechanism to handle asynchronous events. The required classes represent events that can happen and the code to execute when these events occur. This code is scheduled and dispatched by a real-time scheduler.

5) Asynchronous transfer of control

The RTSJ includes a mechanism to extend the Java exception handling to allow applications to change the control between threads. It is necessary a safety way to transfer the control and to finish the execution in a normal way. The Java mechanism to stop the execution of a thread is not safe and it is not approved to be used in real-time systems. This is the reason why the RTSJ provides a new mechanism to handle asynchronous events and to transfer control.

6) Physical memory access

The most applications want to do a productive use of the RTSJ accessing to the physical memory. This is due to most events in a real-time system are handled using peripheries and external sensors. The RTSJ defines some classes to make easy these tasks. One class allows developers to access to the physical memory at byte level. Moreover, another class allows to construct objects in physical memory.

III. MULTIPROCESSOR SUPPORT IN JAVA

A. Java Standard Edition

Since the first versions of Java, classes related to the use of threads and synchronization, located in the java.lang.Thread package, have been used to implement concurrency. These primitive methods are only suitable to carry out basic tasks. Java API 5.0 introduced some new features through the java.util.concurrent package [14] to develop more advanced tasks. These features allow taking advantage of the possibilities of actual multiprocessor systems and multicore processors. The most representatives are:

- Objects lock that support locking instructions and simplify the programming of concurrent applications.
- Executors that define a high-level API to release and to handle threads. The class Executor of the java.util.concurrent package provides methods to
handle pools of threads suitable to develop large-scale applications.

- Concurrent collections that help to reduce the need of synchronization between threads in a given application.
- Atomic variables that have features to minimize the synchronization and help to avoid errors related to memory consistency.

Latest developments concerning concurrency and multiprocessor/multicore in Java have been made with the Java API 7. This Java version includes the framework Fork/Join in the JSR-166 package and it provides new mechanisms in order to use all the processing power available in multiprocessor systems to improve the performance of the Java applications. The Fork/Join package is an implementation of the ExecutorService interface. It is specially designed to carry out tasks that can be recursively divided into distributed subtasks. In the same manner than with any implementation of the ExecutorService, the Fork/Join framework distributes the tasks in a pool of threads [15]. The main difference introduced is the preemptive algorithm. Subprocesses that complete their tasks can steal tasks from other subprocesses that are still occupied.

B. OpenMPI Java

An alternative for parallel programming in multiprocessor systems is to use a message passing based standard, such as Message Passing Interface (MPI). The OpenMPI implementation provides a Java interface currently under development and which is only available in nightly distributions [16]. In January 2013, the OpenMPI development community discussed the possibility of including the Java interface to the last release version of the implementation. However, they finally decided that its current state of development is not enough for official distribution.

The OpenMPI interface for Java is based on the original code called mpiJava and developed by the universities of Indiana and Syracuse [17]. From this code, OpenMPI has maintained and completed the interface, although the documentation associated with it [18].

C. RTSJ

First versions of the RTSJ do not provide any solution to the development of parallel applications in multiprocessor systems. The increasing of this kind of systems and the necessity to optimize the use of their resources has motivated the appearance of new interests. Specifically, the community wants to provide a more direct support to multiprocessor systems in the currently versions of the RTSJ.

The last version of the RTSJ, version 1.1., includes several requirements which are mandatory to develop applications in multiprocessor platforms. These requirements are [1]:

- Allowing to assign schedulable objects to a specific processor.
- Making possible the global planning of schedulable objects across different processors.
- Permitting to use protocols of block queue based on spin-based locks to control the access to shared resources.

- Managing external events in Java has an impact which can be restricted to a subset of the available processors. This allows to control where is running the Java code that answers to an interruption.

The RTSJ provides limited support to multiprocessor systems. Moreover, the characteristic of the underline operative system also has an important influence. The version 1.1 of the works out next areas to give a complete support to this kind of systems:

1. Dispatching model.
2. Processor allocation model.
3. Synchronization model.
4. Cost enforcement model.
5. Affinity of external events.

One of the commercial implementations of the RTSJ that offers a multiprocessor version of the virtual machine is JamaicaVM of Aicas [14]. JamaicaVM supports the RTSJ V1.0.2 but includes specific behavior left open by the specification and included in V1.1. Next we analyze the most relevant aspects related with parallelization that Jamaica takes into account [15].

The main characteristic of the multiprocessor version of JamaicaVM is the garbage collector (GC). It is a real-time GC. It can run on a CPU at the same time that other tasks of the application run on other CPU. Furthermore, several CPUs can run the GC in parallel. The GC is able to stop the application during a period of time to release the necessary memory to attend requests from the application.

It is important taking into account the limitations of the multiprocessor version of JamaicaVM to develop applications, such as Java arrays are not being allocated early at the application startup, before the GC starts to recycle memory and these are allocated using a non-contiguous representation producing high costs to access to the array; it does not support the JVM Tool Interface (JVMTI) and the option “-agentlib” does not work neither in the virtual machine nor in the builder; it does not support the class javax.realtime.MonitorControl; and the class com.aicas.jamaica.lang.Debug does not support the methods getMaxRangeSize, getNumberOfFreeRanges, printFreeListStats and createFreeRangeStats.

To develop applications in multiprocessor systems we have to take into account the proper distribution of tasks using the available processors. On this kind of systems, the scheduler can assign any thread to any CPU, by default and respecting the priorities. This flexibility makes performance decreases or jitter increases, i.e., changes in the execution period of events. The main reason of this behavior is the hard cost of changing a thread from a CPU to another. It involves code and date change to the useless cache, delaying the execution. To avoid changing threads between CPUs, it is possible to reduce the option of the scheduler assigning one thread to a specific CPU. The class AffinitySet allows to define in what CPU can be executed a given thread. It is necessary to test what affinity works better in a specific application. Next, we analyze some cases related to how setting the affinity of threads in different situations.
1) Communication through Shared Memory

If some threads are running on the same CPU, communication through shared memory is usually more efficient. This is because threads running on the same CPU can use the cache of the CPU to communicate between them. In contrast, to send data from a CPU to another, these data have to go before to the main memory, which is slower. To fix two threads that have communication in the same or in different CPU have to look for the balance between computation and communication. On one hand, if there is more computation, is better to use different CPUs. On the other hand, if it is more communication, using the same CPU will have better results. The same can occur when two threads do not communicate but write in the same line of cache. This is known as “false sharing”. In Jamaica, this could happen if two threads modify data of the same object, i.e., in the same block.

2) Performance Degradation on Locking

If two threads can only run on the same CPU and want to access to the same monitor, the runtime system will decide efficiently if the monitor is free and is able to be acquired. Considering next scenario:

- Thread A, with high priority, acquires and releases the same monitor repeatedly.
- Thread B, with low priority, acquires and releases the same monitor repeatedly.

This happen if A and B concurrently read a synchronized data structure. Supposing that thread B starts before thread A. A can wait that B releases the monitor at some time. Then, A continues its execution. As thread A has higher priority than thread B, A will never being interrupted by B. If A and B are assigned in the same CPU, B will never able to run until A stops. If thread A releases the monitor and after a while tries to acquire it again, B will wait again.

In contrast, if A and B can run on different CPUs, thread B can run at the same time than thread A, and acquire the monitor when A releases it. In this case, A has to acquire the monitor from B before it can continue. The additional overhead to block and wake up A after that B releases the monitor could be significant.

3) Periodic Threads

Some applications have periodic events that need to occur with high accuracy. In this case, latencies in the cache can appear. Considering the following scenario:

- Thread A, with high priority, runs every 2 ms for 1 ms.
- Thread B, with low priority, runs every 10 ms for 2 ms.

If both threads run on the same CPU, B will use some of the gaps that A leaves. B can also leave gaps. In this case, when thread A starts again, it needs to charge in cache its code and data. This can produce CPU stalls. These stalls only occurs when B run immediately before A. It does not happen after the gaps produced when the CPU is idle. The fact that the stall occurs sometimes but sometimes not appears as a jitter on the thread. This problem can be resolved or reduced assigning threads A and B to different CPUs.

4) Scheduling analysis

The scheduling analysis is a technique used to determine if real time requirements established prior to execute a set of threads or schedulable objects will be satisfied. We can use different algorithms to carry out the scheduling analysis, such as Rate Monotonic Scheduling (RMS). However, if a subset of tasks has any dependency with other tasks of the application, it will be possible to apply a partial analysis to the dependable tasks.

A scenario where threads guarantee deterministic responses while others run tasks with data in the background is an example that shows the usefulness of this approach. The subset of threads in charge of the deterministic response is isolated to a CPU and the rate-monotonic scheduling can be applied to them.

5) Interruption handler of the Operative System

Operative Systems usually join an interruption handler to a specific CPU. Effects over cache described in the second point of this list, performance degradation on locking, also can occur between the handler code and the threads. Then, the jitter can be reduced running the threads of the application in a different CPU than the CPU used by the operative system to handle the interruptions.

IV. EXPERIMENTS/EVALUATION

In order to contrast the theoretical aspects discussed in the previous sections, we have performed the evaluation of the alternatives presented in Section III, Java SE API 7, Java OpenMPI and RTSJ. To do so, we have implemented a basic example as case study. The aim of these tests is to verify the improvement obtained in the execution time of a problem with a multiprocessor version compared with a sequential one. To do that, we have considered a large vector, in our case, with 10,000,000 elements, in which we have performed a random initialization of the content. We have called the problem RandomFill.

The evaluation of the experiments has been conducted on a virtual machine using two and four of the eight available cores on an Intel® Core (TM) i7 920 CPU 3GHz, 1GB RAM processor and GNU/Linux Operating System / x86_64 Debian Linux with kernel 3.11 Jessie-2-amd64 # 1 SMP.

To solve the problem with Java API 7 we have used the classes ForkJoinPool and RecursiveAction of the package java.util.concurrent. Fig. 1 sample the times obtained in 30 iterations with the sequential version, and the parallel version using 2 of the 4 available cores and the 4 available cores. Also Table 1 resumes the times obtained, with average, typical deviation and the worst case execution time (WCET) for each one of the executed tests.

In the Java API 7 execution the parallel version using 4 cores requires approximately a third part of the time that requires the sequential one. We can see an important fluctuation in the run-time of the parallel version with a high relative error of 9%. Furthermore, the WCET is far from the average value that is normal in a non-predictable JVM.

In the case of OpenMPI, we can see in Fig. 2 the results of the parallel processing of the problem using message passing on Java API 7. We have test the problem using 2, 3 and 4
parallel processes in 2 and 4 cores. In this case including a comparison with the sequential time has no sense because the programming model based on message passing is specially designed to use different processes in parallel. However, we can see a worse performance in contrast with the parallel version carried out with Java API 7. In addition, times also have a very high fluctuation with a relative error of around 19% and a WCET far away from the average. It is probably that the OpenMPI implementation presents high latencies because it is not optimized. As we observe in Fig. 2 and in the resume of results included in Table 1, the difference between times obtained with the execution of 2 and 3 processes on 2 and 4 cores is not significant. However, using 4 processes provides a considerable enhancement. Likewise, the solution of the problem with 4 processes in 2 and 4 cores presents significant differences in time. Times obtained with 4 processes and 4 cores highlight with respect the rest of the outlined options tested. In this case the parallelization of the problem is carried out with one process in each core, whereas using only 2 cores we have 2 processes running on each core.

To solve the problem in JamaicaVM we have used the classes ExecutorService, to create a pool of threads, and AffinitySet, to determine the affinity of the threads respect to each available processor, as we can see in the fragment of code included in Fig. 3. We have also used the Executors class and the Callable interface, the latter to implement the class that contains the functionality of the problem, as Fig. 4 shows. The reduction in time obtained with the parallel version which exploits the multiprocessing is very high, as we can see in Fig 5. The parallel version is much more efficient than the sequential version, in all the cases; using only 2 cores or using the 4 available cores. However, the difference between times obtained in the execution with Java SE 7 is very significant, up to three orders of magnitude. This substantial difference in time is due to the lack of optimization using arrays in JamaicaVM, being this aspect highly refined in the Java API 7. As we discussed above, array elements are allocated using noncontiguous memory representation resulting in high cost to access them. Nevertheless, the dispersion of the measures is more controlled in the RTSJ version than in the Java SE 7 version.

V. CONCLUSIONS

In this paper we have analyzed different alternatives to develop applications in multiprocessor systems using the Java language. We conducted an analysis of the tools included in JavaSE, especially in the API 7. The potential of the OpenMPI Java interface to develop parallel applications based on

<table>
<thead>
<tr>
<th></th>
<th>Java SE 7</th>
<th>Open MPI</th>
<th>JamaicaVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>110.77</td>
<td>85.00</td>
<td>117.30</td>
</tr>
<tr>
<td>Average (ms)</td>
<td>60.00</td>
<td>123,50</td>
<td>121.00</td>
</tr>
<tr>
<td>Standard Deviation (ms)</td>
<td>34.00</td>
<td>68.33</td>
<td>6361.23</td>
</tr>
<tr>
<td>WCET (ms)</td>
<td>117.60</td>
<td>123.47</td>
<td>98.00</td>
</tr>
<tr>
<td></td>
<td>6,51</td>
<td>5.05</td>
<td>5.13</td>
</tr>
<tr>
<td></td>
<td>2.79</td>
<td>6,25</td>
<td>6.05</td>
</tr>
<tr>
<td></td>
<td>3.33</td>
<td>1,24</td>
<td>6.20</td>
</tr>
<tr>
<td></td>
<td>2.13</td>
<td>5,13</td>
<td>63.64</td>
</tr>
<tr>
<td></td>
<td>5.05</td>
<td>6.05</td>
<td>27.45</td>
</tr>
<tr>
<td></td>
<td>6,25</td>
<td>6.20</td>
<td>27.45</td>
</tr>
</tbody>
</table>
message passing has revised, as well. Furthermore, we performed a study of the RTSJ and the enhancements adopted in version 1.1 for the optimal use of resources in this type of systems. We have analyzed the functionality and limitations of multicore support in JamaicaVM, a commercial implementation of the RTSJ. After the theoretical studio of these options, an empirical evaluation has been carried out implementing a parallelizable problem with each one of these alternatives. The execution times obtained in a parallel version of the problem running on a multiprocessor system have been compared with the times obtained with a sequential version. The enhancement of the performance contributed by the parallelism in multiprocessor systems is manifest. Nevertheless, a performance lack has been detected in JamaicaVM using arrays that we want to compare with other implementations of the RTSJ. We are also interested in using multiprocessing algorithms in new generation embedded systems with more than a processor.

**REFERENCES**


