Efficient Execution of Parallel Java Applications

Jordi Guitart, Xavier Martorell, Jordi Torres and Eduard Ayguadé

European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department,

Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6,

E-08034 Barcelona (Spain)

{jguitart, xavim, torres, eduard}@cepba.upc.es

ABSTRACT

In this paper we propose mechanisms to improve the performance of parallel Java applications. The proposal is based on the establishment of a dialog between each Java application and the underlying operating system. This dialog implies modifications at the application (or compiler), the threading library and kernel levels. The paper includes some preliminary experimental results that show how a good cooperation between the Java applications and the execution environment can improve the performance of each individual application.

1. INTRODUCTION

Over the last few years, Java has emerged as an interesting language for the network programming community. This has largely occurred as a direct consequence of the design of the Java language. This design includes, among others, important aspects such as portability and architecture neutrality of Java code, and its multithreading facilities. The latter is achieved through built-in support for threads in the language definition. The Java library provides the Thread class definition, and Java runtimes provide support for thread and monitor primitives. These characteristics, besides others like its familiarity (due to its resemblance with C/C++), its robustness, security and distributed nature also make it a potentially interesting language for parallel scientific computing.

The use of Java for high-performance scientific applications faces a number of problems that are currently subject of research. The first one is related with the large overheads incurred by the interpretation/JIT compilation process and the inefficiencies of the Java implementation, which favor equivalent versions of the same application written in other languages such as Fortran or C. The second problem is related with the lack of programming models for the specification of parallelism. This makes difficult the parallelization of Java applications because it implies the explicit management of the parallelism/synchronization. And third, the threading support available in the JVM incurs large overheads that can easily offset any gain due to parallel execution. Recent proposals [1, 5] propose language extensions and runtime support to ease the specification of Java parallel applications and their efficient execution.

In this paper we analyze causes of performance degradation when parallel Java applications are executed in a multiprogrammed environment. The two main issues that lead to this degradation can be summarized as follows:

- The Java runtime environment does not allow applications to have control on how Java threads map onto kernel threads and to make suggestions about the scheduling of these kernel threads.
- The Java runtime environment does not inform the application about the dynamic status of the underlying system so that the application cannot adapt its execution to these characteristics.

In order to support these two assertions, we have designed an experiment based on LUAppl, a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [5] that uses a matrix of 1000x1000 elements. The experiment consists in a set of executions of LUAppl running with different number of Java threads and kernel threads. Table 1 shows the average results obtained executing in a SGI Origin 2000 architecture [8] with MIPS R10000 processors at 250 MHz running JVM version Sun Java 1.2.2. The first and second rows show that when the number of Java threads matches the number of kernel threads, the application scales quite well. However, if the number of kernel threads provided to support the execution does not match, as shown in the third row, the performance is degraded. In this case the kernel is providing only three threads, probably because the application or the threading system has not appropriately informed the kernel about the concurrency level. This results in an execution time worse than the one achieved if the application would have known that only three kernel threads were available and would have adapted its behavior to simply generate three Java threads.

Java Threads	Kernel Threads	Execution Time in sec
3	3	39.7
4	4	34.3
4	3	44.1

Table 1. LUAppl performance degradation

In this paper we approach the problem in two different ways. In the first one, we propose to use one of the services supplied by the Java native underlying threads library to inform it about the concurrency level of the application. In the second one, we propose to execute Java applications on top of JNE (Java Nanos Environment built around the NanoThreads environment [4]). JNE provides a solid investigation platform to improve the

performance of parallel Java applications on parallel architectures. This is the main objective of the *Barcelona Java Suite* (BJS) research environment, currently under development at CEPBA.

The rest of the paper is organized as follows: Section 2 presents the concurrency level approach. Section 3 presents an overview of the JNE and Section 4 reveals some details of its implementation. Section 5 presents the evaluation of our proposal, and finally, Section 6 presents the conclusions of this paper.

2. CONCURRENCY LEVEL

Current implementations of JVM do not inform the underlying threads layer about the desired concurrency level of the application. By default, the threads library adjusts the level of concurrency itself as the application run using metrics that include the number of user context switches and CPU bandwidth. In order to provide the library with a more accurate hint about the concurrency level of the application, the programmer could invoke, at appropriate points in the Java application, the `pthread_setconcurrency(level)` service of the Pthreads library[1] [3]. The argument `level` is used by Pthreads to compute the ideal number of kernel threads required to schedule the available Java threads.

This approach solves one of the problems mentioned in Section 1. Applications have more control on how Java threads map onto kernel threads, specifying the number of processors on which the application wants to run at any moment. However, other problems remain unresolved. For instance, it does not allow applications to decide about the scheduling of kernel threads. Although it allows to the applications to inform to the execution environment about their processor requirements, it doesn't allow to the execution environment to tell the application about the number of processors assigned to them. In other words, the communication path only exists in one direction. As a consequence, applications cannot react and adapt their behavior to the decisions taken by the underlying system. If informed, applications would be able to restrict themselves in terms of parallelism generation, thus avoiding unnecessary overheads.

3. JAVA NANOS ENVIRONMENT

JNE (Java Nanos Environment) will provide additional mechanisms to improve the communication between the applications and the execution environment, thus allowing applications to collaborate in the thread management.

3.1 Adaptive Java applications

The first issue considered in JNE is the capability of Java applications to adapt their behavior to the amount of resources offered by the execution environment [2]. The process is dynamic and implies three important aspects:

- First, the application should be able to request and release processors at any time. This requires from the execution environment an interface to set the number of processors the application wants to run.

[1] Our implementation targets SGI Origin JVM, which like many other (Linux, Solaris, Alpha, IBM), implements the native threads model using Pthreads.

```
class workerTh extends Thread
{
 LUAppl target;
 int me;
 public workerTh(LUAppl t, int m) {
  target = t;
  me = m;
 }
 public void run() {
  target.parallelLoop(me);
 }
}

class LUAppl {
  .
  .
  .
 public void go() {
  for (k=0;k<SIZE;k++) {
   for (int i=k+1;i<SIZE;i++) {
    mat[i][k] = mat[i][k] / mat[k][k];
   }
   thNum = jne.cpus_current();
   workerTh threads[] = new workerTh[thNum];
   for (int th=0;th<thNum;th++) {
    threads[th] = new workerTh(this,th);
    threads[th].start();
   }
   for (int th=0;th<thNum;th++) {
    try {
     threads[th].join();
    } catch (Exception e) {}
   }
  }
 }

 void parallelLoop (int me) {
  int chunk = (((SIZE)-(k+1))/thNum);
  int rest = ((SIZE)-(k+1))-chunk*thNum;
  int down = (k+1)+chunk*me;
  int up = down+chunk;
  if (me == thNum-1)
   up += rest;
  for (int i=down;i<up;i++) {
   for (int j=k+1;j<SIZE;j++) {
    mat[i][j]=mat[i][j]-mat[i][k]*mat[k][j];
   }
  }
 }

 public static void main (String args[]) {
  // Read NumThreads from properties
  jne.cpus_request(NumThreads);
  SIZE = new Integer(args[0]).intValue();
  new LUAppl().go();
 }
}
```

Figure 1. LUAppl code

- Second, the amount of parallelism that the application will generate (at a given moment) is limited by both the number of processors assigned to the application and the amount of work pending to be executed. The execution environment has to provide an interface to allow the application to check the number of processors available just before spawning parallelism.
- And third, the application should be able to react to processor preemptions and allocations resulting from the operating system allocation decisions. This also requires mechanisms that allow the application, once informed, to recover from preemptions.

3.2 Application/JNE interface

Each Java application executing on the JNE shares information with the execution environment. The information includes the number of processors on which the application wants to run at any moment and the number of processors currently allocated by the operating system to the application.

We have defined a Java class called *jne*, which contains the following two Java methods for calling, through JNI [9], the JNE services for requesting and consulting processors:

- *cpus_current*: method for consulting the number of processors allocated to the calling application when it makes the call.
- *cpus_request(num)*: method for requesting to the operating system num processors.

3.3 JNE scheduler

The JNE scheduler is responsible for the distribution of processors to applications. At any time, there is a current active scheduling policy that is applied to all applications running in the system. The scheduler observes application demands, estimates the load of the machine, and finally distributes processors accordingly. The scheduler also decides which processors are assigned to each application taking into account data affinity issues (i.e. trying to exploit data locality whenever possible).

JNE offers a set of scheduling policies, including batch, round robin, equipartition and others than combine space- and time-sharing. In our evaluation we use Dynamic Space Sharing (DSS) [6,7]. In DSS, each application receives a number of processors that is proportional to its request and inversely proportional to the total workload of the system, expressed as the sum of processor requests of all jobs in the system.

3.4 Examples

Figure 1 shows the source code of the LUAppl (presented in section 1.1), highlighting the utilization of the JNE interface services. In this example, the application requests *numThreads* processors when it starts. If desired, it could change this request whenever a change in the parallelism profile exists in the application. After that, every time it has to spawn parallelism (i.e. at the beginning of each k iteration of the loop), the application checks the number of processors currently allocated to it. With this information, it generates work for as many threads as processors available to run.

As part of our research platform, we are using the JOMP [1] compiler. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. It includes a set of directives and library methods. The compiler automates the generation of parallelism from the directives specified by the user.

For our research, we have modified the implementation of the JOMP compiler and runtime library to implement the communication between the application and JNE. When the user inserts an invocation of the `omp_set_numthreads` method or makes use of the `NUM_THREADS` clause, the compiler injects an invocation of `cpus_request`. Then, before each `PARALLEL` construct the compiler injects an invocation of `cpus_current` in order to adapt its behavior to the available resources.

4. IMPLEMENTATION OF JNE

The JVM implementation of SGI, like many others (Linux, Solaris, Alpha, IBM), implements the native threads model using the Pthreads library (Figure 2a). Thus one Java thread maps directly in one pthread, and Pthreads library schedules these pthreads over the kernel threads offered by the operating system.

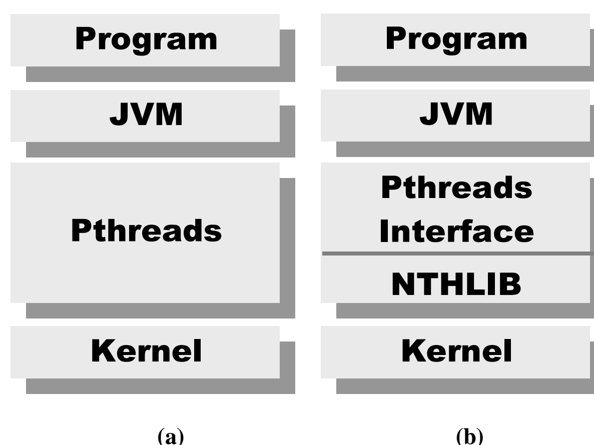

Figure 2. (a) Java IRIX Environment
(b) Java Nanos Environment

In order to implement the mechanisms described in the previous sections we decided to replace the Pthreads library by the Nano-threads Library (NthLib). In order to avoid modifications of the JVM, we maintain the Pthreads library interface but we have rewritten the library methods using the services provided by NthLib (Figure 2b). NthLib is a user level threads package specially designed for supporting parallel applications and establishing the cooperation between application and kernel.

NthLib provides the following services:

- **Thread management services:** nth_create (create a nano-thread), nth_exit (finalize a nano-thread), nth_wait (block a nano-thread) and nth_yield (yield virtual processor to another nano-thread).
- **Generic queue management services:** nth_queue_init (initialize queue), nth_enqueue (enqueue nano-thread on queue) and nth_dequeue (dequeue nano-thread from queue).

- **Ready queue management services:** nth_to_rq_end (enqueue nano-thread on global ready queue) and nth_to_lrq_end (enqueue nano-thread on local ready queue).
- **Mutual exclusion services:** spin_init (initialize spin lock), spin_lock (lock spin) and spin_unlock (unlock spin).

Figure 3 shows a simplified example showing the replacement of the `pthread_cond_signal` service.

```
int pthread_cond_signal(pthread_cond_t *c)
{
 struct nth_desc *nth_id;

 /* dequeue first thread waiting in
     condition variable */
 nth_dequeue(cond_queue(c),nth_id);

 /* add dequeued thread to ready queue */

 if(nth_id != NULL)

  nth_to_rq_end(nth_id);

 return 0;
}
```

Figure 3. Replacement example

5. EVALUATION

In this section, we highlight some conclusions drawn from our initial experimentation with JNE. The current implementation targets the SGI Origin 2000 architecture [8] with MIPS R10000 processors at 250 MHz running the JVM version Sun Java 1.2.2.

We have designed several experiments based on the application LUAppl using a matrix of 1000x1000 elements. First the application is executed in a dedicated environment, with different levels of concurrency (4, 8 and 16 threads). Second, the application is executed as part of a workload composed of 3 LUAppl with a concurrency level of 8 threads each, and 1 LUAppl with a concurrency level of 16 threads, executing in 32 processors with a DSS scheduling policy [6].

Threads	IRIX	IRIX+SETC	JNE
4	67.1	33.1 (50.6%)	28.4 (57.6%)
8	68.3	24.3 (64.4%)	23.2 (66.0%)
16	36.1	23.9 (33.7%)	20.7 (42.6%)

Table 2. LUAppl standalone execution time in sec

5.1 Single application performance

Table 2 shows the average execution time for an individual instance of the LUAppl with different levels of concurrency: 4, 8 and 16 threads. For each experiment, 10 executions have been performed. The first column (labeled IRIX) corresponds to the execution on the original IRIX system, the second column (labeled IRIX+SETC) corresponds to the original IRIX system when the application informs to the Pthreads library about the concurrency level of the application using the `pthread_setconcurrency(level)` call described in section 2, and the third column (labeled JNE) corresponds to the execution time on top of the Java Nanos Environment. Notice that, in the worst case, IRIX+SETC improves the performance in 33.7% while JNE improves in 42.6%.

5.2 Multiprogrammed workloads

In the first workload, three instances of LUAppl with concurrency level of 8 threads and one instance with concurrency level of 16 threads are simultaneously started on 32 processors. In this case, the system is overloaded with a maximum load of 40. Table 3 shows the execution time of the workload. Notice that IRIX+SETC improves the performance in 49.3% while JNE improves the performance in 63.2%.

IRIX	IRIX+SETC	JNE
79	40 (49.3%)	29 (63.2%)

Table 3. Execution time for the first workload in sec

In order to see the impact on each individual instance, we have executed a second workload in which each LUAppl restarts until it reaches 10 repetitions. So in total, 30 instances with concurrency level of 8 threads and 10 instances with concurrency level of 16 threads are executed. Table 4 shows the execution time for this workload. Notice that IRIX+SETC improves the performance in 46.2% while JNE improves the performance in 55.2%.

IRIX	IRIX+SETC	JNE
740	398 (46.2%)	331 (55.2%)

Table 4. Execution time for the second workload in sec

Table 5 shows the average execution time for each application instance. Notice that IRIX+SETC improves the performance of 8-threads instances in 52.6% and of 16-threads instances in 21.8%. JNE improves the performance of the 8-threads instances in 59.6% and of the 16-threads instances in 51.0%. The execution on top of JNE noticeably improves the performance of 16-threaded applications.

Threads	IRIX	IRIX+SETC	JNE
8	68.4 ± 16.0	32.4 ± 2.5 (52.6%)	27.6 ± 5.1 (59.6%)
16	47.2 ± 11.3	36.9 ± 6.3 (21.8%)	23.1 ± 5.5 (51.0%)

Table 5. Execution time (average and stdev) of each application instance in the second workload in sec

Table 6 summarizes the performance degradation of the 8-threaded and 16-threaded instances when executed on a multiprogrammed workload with respect to their standalone execution. Performance degradation is calculated dividing the best application standalone execution time (obtained from column JNE in Table 2) by the average execution time of an application instance in a workload. Notice that in general, JNE minimizes the degradation when applications are executed in a loaded system.

Threads	IRIX	IRIX+SETC	JNE
8	0.34	0.71	0.84
16	0.44	0.56	0.89

Table 6. Performance degradation of the multiprogrammed execution vs. the best standalone execution

6. CONCLUSIONS

In this paper we have shown how the interaction between the application and the underlying resource manager (kernel) may improve the performance of parallel Java applications.

The paper shows two scenarios. In the first one the application is able to inform the underlying execution environment about its concurrency level. Our preliminary experimental results show that this communication improves the behavior either when the application is executed in standalone way or when executed as part of a multiprogrammed workload.

In the second scenario, in addition to this communication path, the kernel is also able to inform the application about the resource allocation decisions. The application is able to react to these decisions, changing the degree of parallelism that it is actually exploited from the application. Again, our preliminary results show a noticeable impact on the final performance.

Both mechanisms are implemented in the *Java Nanos Environment* (JNE), part of the *Barcelona Java Suite* (BJS) research environment, currently under development at CEPBA. JNE provides a solid investigation platform to improve the performance of parallel Java applications on parallel architectures. This is the main objective.

7. ACKNOWLEDGMENTS

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our experiments. This work is supported by the Ministry of Education of Spain under contract TIC98-511 and by Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND.

8. REFERENCES

[1] M. Bull and M.E. Kambites. “JOMP -- an OpenMP-like interface for Java”. Proceedings of the 2000 ACM Java Grande Conference, pp. 44-53. June 2000.

[2] D. Feitelson. “Job Scheduling in Multiprogrammed Parallel Systems”. IBM Research Report 19790, Aug. 1997.

[3] J. Guitart, J. Torres, E. Ayguadé and J. Labarta. “Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications”. 2nd Workshop on Java for High Performance Computing, Santa Fe, New Mexico (USA), pp. 15-25. May 2000.

[4] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé. “A Library Implementation of the Nano Threads Programming Model”. 2nd EuroPar Conference, Lyon, France, pp. 644-649. August. 1996.

[5] J. Oliver, E. Ayguadé, N. Navarro, J. Guitart, and J. Torres. “Strategies for efficient exploitation of loop-level parallelism in Java”. To be published "Concurrency: Practice and Experience", 2001.

[6] E.D. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T.S. Papatheodorou and N. Navarro. “Kernel–level Scheduling for the Nano-Threads Programming Model”. 12th ACM International Conference on Supercomputing (ICS’98), Melbourne, Australia, pp. 337-344. July 1998.

[7] E.D. Polychronopoulos, D.S. Nikolopoulos, T.S. Papatheodorou, X. Martorell, J. Labarta and N. Navarro. “An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors”. 12th International Conference on Parallel and Distributed Computing Systems (PDCS’99), Fort Lauderdale (Florida - USA), pp. 148-155. August 18-20, 1999.

[8] Silicon Graphics Inc. Origin200 and Origin2000 Technical Report, 1996.

[9] Sun Microsystems. “Java Native Interface”. March 2000. http://java.sun.com/products/jdk/1.3/docs/guide/jni/index.html