Improving Java Multithreading Facilities: the *Java Nanos Environment*

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 will present the Java Nanos Environment (JNE), an execution environment that shows the advantage of establishing dialog between Java applications and the underlying operating system kernel. We will describe the characteristics of the kernel-level scheduling mechanism proposed and the results we have achieved. From the results we show that a good cooperation between the Java application and the multithreading runtime could speedup the application execution time. In this paper we will show that Java presents an important opportunity for the HPC community but that certain adjustments in the specification of the language that allows to consider the interaction between the runtime and the application will be crucial to its widespread use for numerical programming.

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, monitor and condition variable 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.

However, the use of Java for scientific parallel programming has to face the large overheads caused by the interpretation of the bytecode that lead to unacceptable performances. Many current JVMs try to reduce this overhead by Just-in-Time compilation. This mechanism tries to compile JVM bytecodes into architecture-specific machine code at runtime (on the fly). Static Java compilers such as the one produced by IBM [7] have achieved 70-100% of optimized native code. In any case, the naive use of the threads support provided by Java may incur overheads that can easily offset the gain due to parallel execution.

Nowadays, programming in Java over parallel architectures has several limitations that advise against it use due to the low performance obtained. These limitations can be resumed in:

- Java environment doesn't allow to the applications to have control on how Java threads maps onto kernel threads, or to decide about the scheduling of kernel threads.
- Java environment doesn't inform the application about the characteristics of the underlying multiprocessor architecture, thus the application cannot adapt its execution to these characteristics.
- Propose alternatives to resolve these problems become a hard work, due to the lack of flexibility of the Java environment.

In this paper we are going to present two approaches in order to resolve these problems.

In the first approach, we propose to use services supplied by the Java underlying threads library to inform this library about the concurrency level of the application.

In the second approach, we propose to execute Java applications in the NANOS environment [6], creating the Java Nanos Environment (JNE), which will allow us to have a solid investigation platform to improve the performance of parallel Java applications over parallel architectures. This is the final objective that targets the development of *Barcelona Java Suite* (BJS), which is being currently carried out at CEPBA [1].

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. In order to give the library a hint about the concurrency level of the application, we have made experiments introducing an invocation to the `pthread_setconcurrency(level)` service of the threads library. Argument `level` is used to compute the ideal number of kernel threads for scheduling the available Java threads.

These experiments have revealed that inform to the threads library about the concurrency level of the application has an important incidence on the performance. The improvements that we have experimented range from 23% to 58%. Detailed explanation of the effects on performance of the concurrency level can be found at [3].

This approach resolves some of the drawbacks appeared when Java is used in scientific parallel computation, for example:

- It allows to the applications to have control on how Java threads maps onto kernel threads, specifying the number of processors on which the application wants to run at any moment.

But there are others that remain unresolved:

- It doesn't allow to the applications to decide about the scheduling of kernel threads. It depends on the underlying threads library.
- Although it allows to the applications to inform to the operating system about their processor requirements, it doesn't allow to the operating system to answer to applications with the number of processors assigned to them at any moment. Therefore the communication between the applications and the operating system is only one-way and the applications cannot adapt to the available resources, decreasing performance.

In order to resolve these limitations, we propose another approach, called Java Nanos Environment, which increases the communication between the applications and the operating system, allowing to the applications to collaborate in the thread management.

3. JAVA NANOS ENVIRONMENT

In this section, we are going to describe the Java Nanos Environment (JNE) as an execution environment for doing research on performance improvement of parallel Java applications.

3.1 Adaptability to the available resources of Java applications

We propose parallel Java applications whose execution is able to adapt [2] to changes in the number of processors assigned to it. The adaptation is dynamic, at run time, and includes three important aspects: first, the amount of parallelism that the application generates at any time is limited someway by both the number of processors assigned to the application and the current amount of work already pending to be executed. Second, the application is able to request and release processors at any time. And third, the application should be able to adapt to processor preemptions and allocations resulting from the operating system allocation decisions. With respect to the first aspect, the Java thread starting the execution of a parallel region takes the decision about how many processors to use for spawning the parallelism. The operating system has to provide some interface to allow the application to check which is the number of processors available for spawning parallelism. By checking the number just before spawning parallelism, the application ensures that it is going to use the processors currently allocated to it. The second aspect, enabling the request for processors, requires from the operating system interface to set the number of processors each application wants to run on. The operating system should guarantee that the number of requested processors from each application is considered as soon as it distributes processors among applications. The third aspect, applications being able to adapt to processor preemptions, requires also some help from the operating system. The operating system moves processors from one application to another following some scheduling policy (e.g., time sharing). The requirement from the application point of view is that preemptions must not occur. As this is usually not possible, the runtime execution environment may help to provide such a feeling, by recovering preemptions. A good solution from the operating system point of view is, on one hand, to provide some mechanism to reduce preemptions at a minimum, and on the other hand, to provide a complete interface for preemption recovery.

3.2 Kernel-level scheduling

Kernel-level scheduling consists of a set of policies to distribute processors to applications. At any time, there is a current active scheduling policy, applied to all applications running in the system. The active policy can be dynamically changed without incurring any overhead to the running applications. Applications notice only the performance differences obtained from the processor allocation decisions taken by the policy newly established. Different application workloads can benefit from different policies. The active scheduling policy is in charge of looking at the requirements of all running applications and deciding which resources to allocate to each one. Each parallel application is considered as a whole. As long as the policy decides to allocate a number of processors to each application, a portion of the machine is effectively given to that application and the application decides what to do with the processors. The mechanism in charge of determining the exact processors to be assigned to each application ensures that the processors assigned to the application are going to be the ones that more recently have been running on it, if any, thus enforcing data locality. Specific architectural characteristics, such as a NUMA memory subsystem can also be taken into account at that point. The benefit of looking at applications as a whole is that in short term scheduling decisions, processors know where to look first for work (the application where they are assigned to). In case the application has no work to perform, its cooperation with the operating system makes it to release some processors, which will search for work in other applications. The scheduling policies used in this work are presented in Section 2.2.4.

3.2.1 Sharing information with Java applications

Each Java application executing on the JNE shares information with the operating system. The information dynamically flows from the application to the operating system and vice versa. The information includes, but is not limited to, 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. From the number of requested processors, the operating system decides, in a first step, how many processors to allocate to each application. Processors are moved, in a second step, from one application to another. Each moved processor leaves a virtual processor preempted in the source application. It is possible that between the two steps, some time passes to allow the application to voluntarily release the processors to be moved. This functionality is designed to avoid as much as possible the preemption of running processes by the operating system. Along with the number of requested and allocated processors, information about each one of the virtual processors can be checked by the user level execution environment during synchronizations, and help the application progress when the operating system decides to reallocate processors to another application. The best mechanism to inform the user level about preemptions is shared memory, which is asynchronous and does not disturb further the execution of the application. The application does not need to know about preemptions till it reaches a synchronization point, where it can check the shared memory.

3.2.2 Synchronization and processor preemptions

Each time an application needs to do some operation which depends on the number of running processors, it uses the number of processors allocated provided by the JNE interface. This ensures that, at least during a short amount of time, such processors are available (in average, during half a scheduling period or quantum). This means that most of the times, threads are not going to lose a synchronization point, so they are not going to delay the whole application execution. In these situations, the behavior of the application depends, during a certain amount of time, on the number of processors allocated. Typically, this happens when the application spawns parallelism, checking the number of processors allocated to know how many processors are going to participate in the parallelism. From that point to the next synchronization point, in a barrier, or while joining the parallelism, the processors should remain assigned to the application, avoiding that a delay in the synchronization slows down the execution of the application. If the operating system decides to reallocate some processors during the execution of the parallelism, some of the virtual processors will be preempted. This can occur, and the JNE user level execution environment will be always informed, thus detecting the preemptions when reaching the next synchronization point. No time will be lost waiting for synchronization with a preempted processor. Also, when preemption is detected, any processor of the application (usually the one that detects the preemption) can be directly transferred to execute the preempted work, recovering the application progress.

3.2.3 The Java application as the scheduling target

The JNE distributes processors among the running applications, having into account the applications as a whole and their exact requests. Looking at the requests of all the running applications, along with their priorities, the operating system can figure out which is the load of the machine, which applications have more priority to be executed and it can distribute processors accordingly. To minimize movements of processors between applications, a processor allocated to an application searches for work in that application first. In case there is no ready virtual processor to run in its application, the processor is allowed to automatically assign itself to another application and get work from it. Usually, the scheduling policy applied at each quantum prepares a list of applications that have been given fewer processors than requested. Those applications are the candidates to receive the processors that become free due to some application termination. The scheduling policies implemented on the JNE range from the well known equipartition, batch or round robin policies to other kind of policies that can make more use of the information provided by the applications. They are described in next section.

3.2.4 Kernel-level scheduling policies

In this section we summarize the characteristics of the scheduling policies used in the evaluation of JNE. Dynamic Space Sharing (DSS) [9,10] is a two–level scheduling policy. The high level space–shares the machine resources among the applications running in the system. The low level improves the memory performance of each program by enforcing the affinity of kernel threads to specific physical processors.

DSS distributes processors as evenly as possible among applications taking into account the full workload of the system and the number of processors requested by each application. 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.

3.3 JNE advantages

Previous explanation proves that JNE is able to solve many of the drawbacks appeared when Java is used in scientific parallel computation. Concretely, next topics are resolved by JNE:

- JNE allows to the applications to have control on how Java threads maps onto kernel threads, specifying the number of processors on which the application wants to run at any moment.
- JNE allows to the applications to decide about the scheduling of kernel threads, specifying one of the policies supplied by JNE.
- JNE allows to the applications to inform to the operating system about their processor requirements, as well as, JNE allows to the operating system to answer to applications with the number of processors assigned to them at any moment. Thus JNE supplies a two-way communication that allows to applications to adapt to the available resources, increasing performance.

4. IMPLEMENTATION

This section describes some relevant details of the JNE implementation.

4.1 The Nano-threads library

The Nano-threads Library, NTHLIB is a user level threads package specially designed for supporting parallel applications. The role of NTHLIB is two fold. On one hand, NTHLIB provides the user level execution environment in which applications execute. On the other hand, NTHLIB cooperates with the operating system level. NTHLIB and the operating system cooperate by interchanging significant fine grain information on accurate machine state and resource utilization, throughout the execution of the parallel application. When the load of the system is high, each application should run as if executing in a smaller (dedicated) machine. In this case, resource sharing is unavoidable and usually prevents achieving a performance comparable to the individual applications execution, due to conflicts in processors, memory, etc. Supplying accurate information to the operating system about resource needs is a key aspect for getting a good compromise between time and space sharing. When the application shrinks its parallelism, processors could become quickly available for other applications running in the system, reducing idle time. On the other hand, when an application needs more processors, it is guaranteed that the request will be taken into account in a short enough amount of time.

4.2 NTHLIB services

In this section we describe the services supplied by NTHLIB.

- Thread management services
  - ✓ nth_create.- Create a nano-thread.
  - ✓ nth_exit.- Finalize a nano-thread.
  - ✓ nth_wait.- Block a nano-thread.
  - ✓ nth_yield.- Yield virtual processor to another nano-thread.
- Queue management services
  - ✓ nth_queue_init.- Init queue.
  - ✓ nth_enqueue.- Enqueue nano-thread on queue.
  - ✓ nth_dequeue.- Dequeue nano-thread of queue.
  - ✓ nth_to_rq_end.- Enqueue nano-thread on global ready queue.
  - ✓ nth_to_lrq_end.- Enqueue nano-thread on local ready queue.
- Mutual exclusion services
  - ✓ spin_init.- Init spin.
  - ✓ spin_lock.- Lock spin.
  - ✓ spin_unlock.- Unlock spin.

4.3 Threads library replacement

In this section, we are going to introduce the basics of how to execute Java applications in the NANOS environment.

Most of current implementations of the JVM allow Java threads to be scheduled by the virtual machine itself (the so-called *green threads model*) or by the operating system (the so-called *native threads model*). When using green threads, the operating system does not know anything about threads that are handled by the virtual machine. From the point of view of the operating system, there is a single process and a single thread; it is up to the virtual machine to handle all the details of the threading API. In the native threads model, threads are scheduled by the operating system that is hosting the virtual machine. This model must be used to execute parallel Java applications, because otherwise, applications won't be able to exploit architecture possibilities.

The JVM implementation of SGI, like many others, implements the native threads model using the Pthreads library (Figure 1a). Thus one Java thread maps directly in one pthread, and Pthreads library schedules these pthreads over the kernel threads offered by the operating system.

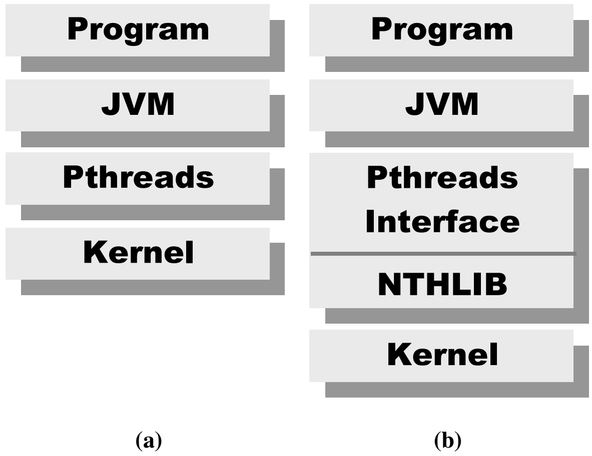

Figure 1. (a) Java IRIX Environment
(b) Java Nanos Environment

In order to execute Java applications in the NANOS environment, we have to replace Pthreads library with NTHLIB library, but we want to avoid the modification of JVM code, thus the JVM will remain calling Pthreads library. But we are going to rewrite Pthreads library, keeping the same interface, so it will offer the same services that the old one, but it will be implemented using the NTHLIB services to provide its functionality (Figure 1b). Next simplified example shows the replacement mechanism:

```
int pthread_cond_signal(pthread_cond_t *c) {
 struct nth_desc *nth_id;
 /* dequeue first thread waiting in
     condition variable */
 nth_dequeue_i(cond_queue(c),nth_id);
 /* add dequeued thread to ready queue */
 if(nth_id != NULL)
  nth_to_rq_end_i(nth_id);
 return 0;
}
```

4.4 Sharing information mechanism between Java applications and JNE

Like we have exposed on section 2.2.1, each Java application executing on the JNE shares information with the operating system. 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. In this section we are going to describe the mechanism used to share this information. JNE has two 'C' methods, *nth_cpus_request* and *nth_cpus_current,* for requesting processors to the operating system and consulting the processors allocated to an application respectively.

We have defined a Java class called *BjsNthLib*, which contains the following two Java methods for calling, through JNI [12], the native methods named before:

- *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.

Figure 2 shows the source code of the LUAppl (a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [8]), highlighting the utilization of these services.

```
class LUAppl {
 static int NumThreads=1;
 static int SIZE;
 double mat[][];
 int k;
 int thNum;
 public void go() {
  // init matrix
  mat = new double[SIZE][SIZE];
  int init = 1325;
  for (int i=0;i<SIZE;i++) {
   for (int j=0;j<SIZE;j++) {
    init = (3125 * init)%65536;
    mat[i][j]=(init-32768.0) / 16384.0;
   }
  }
  for (k=0;k<SIZE;k++) {
   for (int i=k+1;i<SIZE;i++) {
    mat[i][k] = mat[i][k] / mat[k][k];
   }
   thNum = BjsNthLib.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[]) {
  static int NumThreads=1;
  String str =
System.getProperty("JAVA_MP_THREADS");
  if (str != null)
   NumThreads = new Integer(str).intValue();
  BjsNthLib.cpus_request(NumThreads);
  SIZE = new Integer(args[0]).intValue();
  new LUAppl().go();
 }
}


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

Figure 2. LUAppl code

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 [11] 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 (a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [8]) for evaluate our proposal.

5.1 Standalone evaluation

In this section, we evaluate the performance of the LUAppl after the replacement of Pthreads library with NTHLIB library. Table 1 shows the results of the execution of LUAppl of matrix size 1000x1000 and different concurrency levels (4, 8, 16) with original IRIX JVM, with IRIX JVM after informing Pthreads library about the concurrency level of the application, and JVM of the JNE. We can see that IRIX JVM after setting the concurrency level improves the performance in 24,12% in the worst case, and JNE improves the performance in 36,46% in the worst case.

	IRIX JVM	IRIX JVM + SETCONCURRENCY	JNE
4	48096	32613 (32,19%)	25669 (46,63%)
8	43451	24723 (43,10%)	22393 (48,46%)
16	30504	23147 (24,12%)	19381 (36,46%)

Table 1. LUAppl standalone execution time in ms

5.2 Workload 1

In this section, we have designed a workload that consists of 3 LUAppl with matrix size of 1000x1000 and a concurrency level of 8 threads, and 1 LUAppl of matrix size of 1000x1000 and a concurrency level of 16 threads, executing in 32 processors with a DSS scheduling policy. Table 2 shows the results of the execution of this workload with original IRIX JVM, with IRIX JVM after informing Pthreads library about the concurrency level of the application, and JVM of the JNE. We can see that IRIX JVM after setting the concurrency level improves the performance in 49,37%, and JNE improves the performance in 63,29%.

IRIX JVM	IRIX JVM + SETCONCURRENCY	JNE
79000	40000 (49,37%)	29000 (63,29%)

Table 2. Workload 1 global execution time in ms

5.3 Workload 2

In this section, we have designed a workload that consists of 3 LUAppl with matrix size of 1000x1000 and a concurrency level of 8 threads, and 1 LUAppl of matrix size of 1000x1000 and a concurrency level of 16 threads, executing in 32 processors with a DSS scheduling policy. In this workload, each LUAppl restart until it reaches 10 repetitions. Table 3 shows the results of the execution of this workload with original IRIX JVM, with IRIX JVM after informing Pthreads library about the concurrency level of the application, and JVM of the JNE. We can see that IRIX JVM after setting the concurrency level improves the performance in 46,27%, and JNE improves the performance in 55,27%.

JVM IRIX	JVM IRIX + SETCONCURRENCY	JNE
740000	398000 (46,27%)	331000 (55,27%)

Table 3. Workload 2 global execution time in ms

Table 4 shows the average execution time for each application instance. We can see that IRIX JVM after setting the concurrency level improves the performance of 8-threaded applications in 52,53% and on 16-threaded applications in 21,64%, and JNE improves the performance of 8-threaded applications in 59,64% and on 16-threaded applications in 50,94%. Note that JNE increments the performance especially of 16-threaded applications.

	JVM IRIX	JVM IRIX + SETCONCURRENCY	JNE
8	68419,50 ± 16064,71	32478,97 ± 2543,26 (52,53%)	27611,90 ± 5122,57 (59,64%)
16	47207,20 ± 11328,74	36993,40 ± 6303,45 (21,64%)	23160,80 ± 5566,31 (50,94%)

Table 4. Workload 2 execution time per application instance in ms

5.4 Workload 3

In this section, we have designed a workload that consists of 3 LUAppl with matrix size of 1000x1000 and a concurrency level of 8 threads, and 1 LUAppl of matrix size of 1000x1000 and a concurrency level of 16 threads, executing in 32 processors with a DSS scheduling policy. In this workload, each LUAppl restart until one of them reaches 10 repetitions. Table 5 shows the results of the execution of this workload with original IRIX JVM, with IRIX JVM after informing Pthreads library about the concurrency level of the application, and JVM of the JNE. We can see that IRIX JVM after setting the concurrency level improves the performance in 27,29%, and JNE improves the performance in 38,54%.

JVM IRIX	JVM IRIX + SETCONCURRENCY	JNE
480000	349000 (27,29%)	295000 (38,54%)

Table 5. Workload 3 global execution time in ms

Table 6 shows the average execution time for each application instance. We can see that IRIX JVM after setting the concurrency level improves the performance of 8-threaded applications in 53,00% and on 16-threaded applications only in 1,92%, and JNE improves the performance of 8-threaded applications in 56,36% and on 16-threaded applications in 42,90%. Note that JNE increments the performance especially of 16-threaded applications also in this case.

	JVM IRIX	JVM IRIX + SETCONCURRENCY	JNE
8	66523,40 ± 16732,92	31264,63 ± 1354,16 (53,00%)	29029,50 ± 5683,94 (56,36%)
16	40672,70 ± 6919,47	39891,88 ± 2735,96 (1,92%)	23222,60 ± 1212,66 (42,90%)

Table 6. Workload 3 execution time per application instance in ms

6. CONCLUSIONS

In this paper we have described the Java Nanos Environment (JNE) as an execution environment that shows the advantage of the dialog established between Java applications and the underlying operating system kernel. We have described the characteristics to kernel-level scheduling and the results we have achieved replacing the default threads library (Pthreads) with the NTHLIB without modifying the JVM code and using the NANOS environment.

As can be deduced from the previous results, a good cooperation between the application and the multithreading runtime could speedup the application execution time. But the Java specification does not considers the interaction between the runtime and the application. In this paper we have shown that Java presents an important opportunity for the HPC community but that certain adjustments to the language will be crucial to its widespread use for numerical programming.

7. ACKNOWLEDGMENTS

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our experiments. This work is partially supported by the Ministry of Education of Spain under contract TIC98-511.

8. REFERENCES

[1] European Center for Parallelism of Barcelona (CEPBA), http://www.cepba.upc.es.

[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 (part of the 14th ACM International Conference on Supercomputing ICS'00), Santa Fe, New Mexico (USA). May 7, 2000.

[4] X. Martorell, J. Corbalan, D. S. Nikolopoulos, N. Navarro, E. D. Polychronopoulos, T. S. Papatheodorou and J. Labarta. "A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager" Proc. of the 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2000), in conjunction with the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2000), Cancun (Mexico), May 2000.

[5] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, “Nano-Threads Library Design, Implementation and Evaluation”, Technical Report, Universitat Politècnica de Catalunya, UPC-DAC-1995-33, Sep. 1995.

[6] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, “A Library Implementation of the Nano Threads Programming Model”, Proceedings of the 2nd. Euro Par Conference, Lyon, France, August. 1996.

[7] Samuel P. Midkiff, Jos E. Moreira, Marc Snir “Optimizing Array Reference Checking in Java Programs". IBM Systems Journal 37(3): 409-453 (1998)

[8] J. Oliver, E. Ayguadé and N. Navarro. “Towards an efficient exploitation of loop-level parallelism in Java”. In Proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 2000.

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

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

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

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