Report on behalf of the Marie-Curie Research Training Network FAST

Utilizing Java Concurrent Programming, Multi-Processing and the Java Native Interface

Running Native Code in Separate Parallel Processes

J.W. van der Veen
MRS Core Facility, NIMH, NIH, USA

D. van Ormondt and R. de Beer
Applied Physics, TU Delft, NL
E-mail: r.debeer@tudelft.nl

2012-03-05 15:14

Abstract

In this report we describe a Java-based parallel-computing software system for increasing the computational speed of numerically intensive native C/C++/Fortran 90 code. In this software system we have combined three techniques, being Java Concurrent Programming, multi-processing and the Java Native Interface (JNI).

We have found, that invoking −in concurrent Java threads− native C/C++/Fortran 90 code, that is linked via the Java JNI, may frequently lead to native crashes. This behaviour can be circumvented, however, by running the concurrent threads in separate operating system processes (i.e. by using multi-processing).

Index Terms

Java Concurrent Programming, multi-processing, Java JNI, native C/C++/Fortran 90 code

CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Recently we have reported [1] on utilizing Java Concurrent Programming [2] to execute external native programs in a parallel fashion on multi-core CPUs. To that end the external native programs were scheduled in concurrent Java threads by using the Java `Runtime.exec(command)` method [3].

In this work we elaborate further on this approach by including the Java Native Interface (JNI) [4]. That is to say, native codes now are called via the Java JNI as native C/C++ functions and eventially, in turn, as Fortran subroutines. This has the advantage that computational results can easily be returned to the Java environment. The latter is important when using the approach for the *j*MRUI software package [5].

II. METHODS

Just like in our previous report [1] on Java Concurrent Programming we describe the current method on the hand of a conceptual Unified Modeling Language (UML) [6] class diagram (see Figure 1).

When comparing this class diagram to the corresponding UML class diagram in [1] the following should be noted:

- A new Java class, called `NativeC`, has been added. This class calls −via the Java JNI− the native C function `callNativeC()`. This C function returns a double[] array, which may contain native computational results.
- The `NativeC` class represents a standalone Java application by having its own `main()` method.
- The Java `ParallelTask` class executes `NativeC` in a *separate operating system process* by using the Java `ProcessBuilder.start()` method [7]. This method is directly related to the Java `Runtime.exec()` method [3]. The `ProcessBuilder.start()` method, however, is now the preferred way to start a process, particularly if one wants to modify the operating system environment [8].
- The `CollectResults` class for collecting parallel results (see again [1]) may be less important when using the Java JNI return facilities.

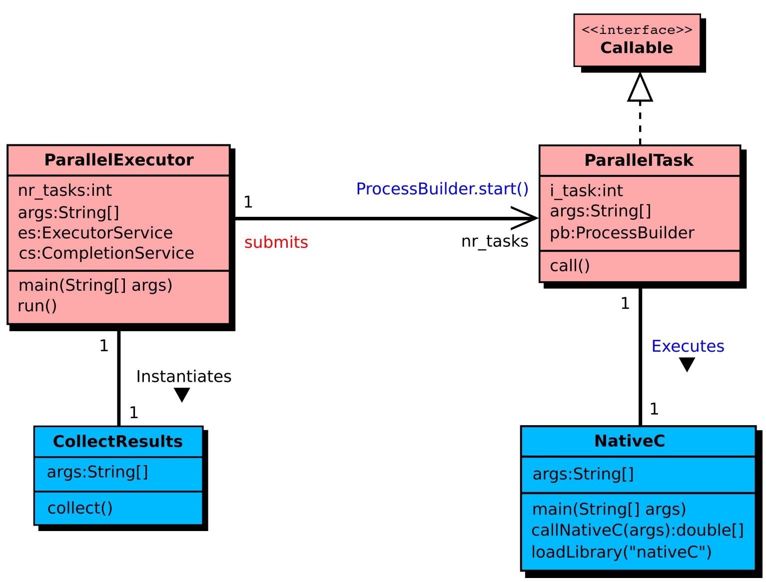

Figure 1. Conceptual UML class diagram of our Java Concurrent Programming approach, when combined with multi-processing and the Java JNI method.

III. RESULTS

In Figure 2 an example of the standard outputs of `ParallelExecutor`, `ParallelTask`, `NativeC` and a genetic algorithm based Fortran 90 subroutine are presented. They were generated by our Java Concurrent Programming software system, described in Figure 1. The functionality of the Fortran 90 subroutine is the same as the external Fortran 90 program, described in our previous report on Java Concurrent Programming (see [1]).

When compared to the standard outputs, displayed in Figure 5 of [1], it is found that the total computational times are about the same. In addition, Figure 2 in this report shows standard outputs, related to returning the results (for the amplitudes) back to the Java environment (the elements of the `dreturn` array). This was realized via suitable Java JNI functions.

```
Hello in run() method of ParallelExecutor!
Time_begin (milli-seconds) = 1330865317304

Number of available processors = 2

Start of task(0):
Start of task(1):
dreturn[0] = 144.4225553333247
dreturn[1] = 544.8420518787989
dreturn[2] = 280.26148476724575
dreturn[3] = 115.91944575196497
dreturn[4] = 394.1423681980491

CALL PIKAIA :
seed = 1234567
RESULTS
     ampl() =    144.4226   544.8421   280.2615   115.9194   394.1424
     freq() =     2.07072    4.05997    0.09451    0.02987   -5.93997
END PIKAIA

Return of task(0) (seconds) = 13.267
dreturn[0] = 144.39175434392962
dreturn[1] = 542.8091306779028
dreturn[2] = 279.19460056626576
dreturn[3] = 117.11698220308591
dreturn[4] = 395.79995509148637

CALL PIKAIA :
seed = 7654321
RESULTS
     ampl() =    144.3918   542.8091   279.1946   117.1170   395.8000
     freq() =     2.08713    4.07800   -0.14992    0.14150   -5.94031
END PIKAIA

Return of task(1) (seconds) = 13.206

Time_end (milli-seconds) = 1330865330615
Total computational time (seconds) = 13.311
```

Figure 2. Standard outputs of `ParallelExecutor`, `ParallelTask`, `NativeC` and a genetic algorithm based Fortran 90 subroutine (see text), while running our Java Concurrent Programming software system on an Intel Core 2 Duo E8400 CPU. Note the standard outputs of the elements of the `dreturn` array (in this example the values of the amplitudes).

IV. Discussion

A. Introduction

When introducing Java JNI in the current work, we at first have omitted the Java `ProcessBuilder.start()` step. That is to say, the `NativeC` class contained no `main()` method and an object of the class was directly instantiated by `ParallelTask` (no creation of a separate operating system process).

It was found that this approach (i.e. parallel processing without using multi-processing) was *highly unstable*. By that we mean, that frequently there were crashes in the native code. These crashes could occur, even when simply restarting a combination of Java and native code that previously had finished the calculation.

Another aspect was, that the occurrence of crashes could be influenced by introducing timing in the Java code (via Java `Thread.sleep(time)`'s) and/or by changing the native computational workload.

In the next subsection we briefly point out how JNI-wrapped native code may be vulnerable to instability [9] [10]. Nevertheless, we sometimes succeeded in carrying out a `ProcessBuilder`-omitted parallel calculation without a native crash, as is shown in Figure 3 for a GammaPress example.

The example of Figure 3 concerns a $17{\times}(1 \times 17)$ calculation of a FID of the myo-inositol metabolite. The total computational time relates well to the one obtained by using the `PPSS` Linux `bash` script [11].

B. Unstable JNI-wrapped native code

In general, a software system consisting of Java and native components, interacting via the Java JNI, may be unsafe [9]. This is because Java is a safe language whereas languages like C, C++ or Fortran 90 are inherently unsafe. For instance, in C/C++/Fortran 90 the memory management is handled by the programmer, which may lead to premature deallocation (dangling pointers) and incomplete deallocation (memory leaks). In Java, on the other hand, the memory management is an automatic process carried out by the Java garbage collector. As a result of unsafe interoperation with the Java code, the native C/C++/Fortran 90 code may become unstable. We have the idea that the latter is *particularly true*, when using *parallel* threads in the Java Concurrent Programming approach.

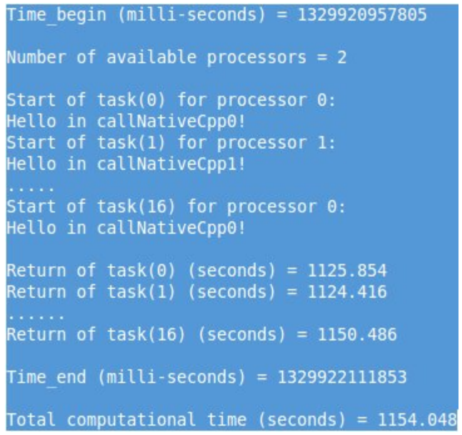

Figure 3. Standard outputs of `ParallelExecutor` and `ParallelTask`, while calling GammaPress as a Java native C++ function on an Intel Core 2 Duo E8400 CPU based desktop PC. In this case multi-processing via the Java `ProcessBuilder` was *omitted* (see text).

In order to test native instability, when applying the `ProcessBuilder`-*omitted* approach, as described in the previous subsection, we have used as native code the very simple Fortran 90 subroutine, shown in Figure 4. Since, as far as we know, there is no means of directly linking Java and Fortran, we have used C as intermediate code.

The essential elements in the code are the local array `local_arr` and the shared array `shared_arr`, which both are `allocate`'d and `deallocate`'d in the code. In addition, there is a waiting time (in this example of 1 s), realized by `call sleep(1)`. The latter was introduced in order to simulate the time of a computational intensive workload.

Besides the waiting time in the Fortran 90 code, there was also a waiting time in the Java code. This second waiting time (realized by the Java `Thread.sleep(time)` method) was introduced in `ParallelExecutor`, when calling `ParallelTask`.

```
!
  module double
  integer, parameter :: dp = kind(0.0d0)
  end module double
!
  module sharing
  use double
  implicit none
  complex(kind=dp), dimension(:), allocatable :: shared_arr ! Shared
  end module sharing
!
  subroutine semipar()
!
  use double
  use sharing
!
  implicit none
  real(kind=dp), dimension(:), allocatable :: local_arr ! Local
!
  write(*,'(''Hello in semipar!'')')
!
  allocate( local_arr(2))
  allocate(shared_arr(2))
!
  call sleep(1) ! To simulate computational intensive workload
!
  deallocate( local_arr)
  deallocate(shared_arr)
!
  return
  end subroutine semipar
!
```

Figure 4. Code of a Fortran 90 subroutine, used to test Java Concurrent Programming with the Java JNI.

In Figure 5 we show the results of this test for three different combinations of the two waiting times. From the figure it can be seen, that a Fortran runtime error occurs for attempting to allocate the already allocated shared array `shared_arr` if the simulated working load (the Fortran `sleep`) is *longer* than the Java `sleep`. This is precisely the situation for carrying out the computational intensive workload in a *parallel* fashion.

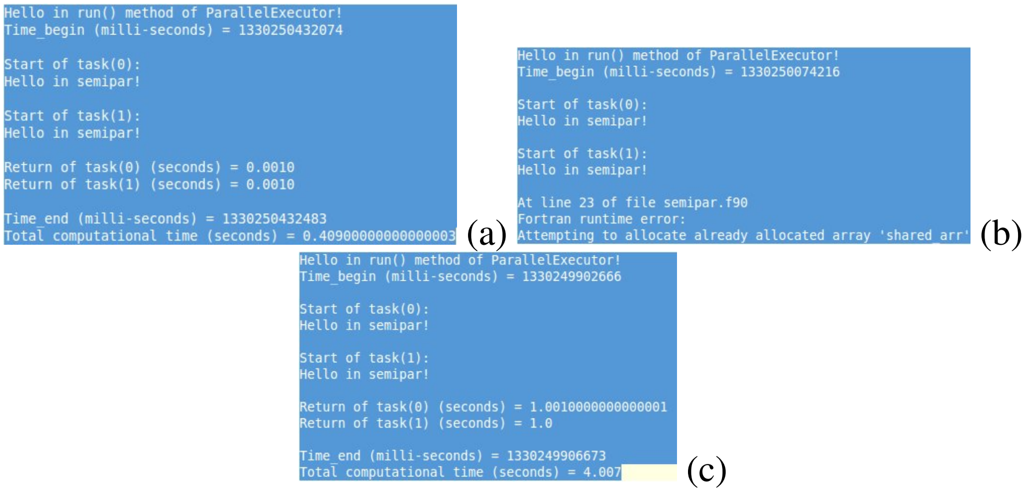

Figure 5. Standard outputs of `ParallelExecutor` and `ParallelTask`, while calling a Fortran 90 test subroutine. (a) Java `sleep` = 0.2 s and Fortran `sleep` = 0 s. (b) Java `sleep` = 0.2 s and Fortran `sleep` = 1 s. (c) Java `sleep` = 2 s and Fortran `sleep` = 1 s (see text).

At the end of this subsection it is important to emphasize, that after introducing in our Java Concurrent Programming approach the Java `ProcessBuilder` class (see again Figure 1), we obtained no further crashes in the native code [10] (as far as we have tested).

V. Summarizing Remarks

Summarizing we like to make the following remarks:

- We have realized a Java-based parallel-computing software system that invokes native C/C++/Fortran 90 code, while *utilizing multi-core* CPUs.
- Unstable behaviour of native code, when linked to *concurrent* Java code via the Java JNI, can be circumvented by applying *multi-processing*.

Acknowledgment

This work was supported by of the Marie-Curie Research Training Network FAST.

References

[1] J.W. van der Veen, R. de Beer and D. van Ormondt, "Utilizing Java Concurrent Programming for Speeding up External Programs. Fully accessing multi-core CPUs by making use of the `java.util.concurrent` package," Report on behalf of the Marie-Curie Research Training Network FAST, 2012, . 3, 4
[2] Oracle, "`Package java.util.concurrent`," http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html, 2012, Utility classes commonly useful in concurrent programming. 3
[3] ——, "`Class Runtime`," http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html, 2012, Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. 3
[4] WikipediA, the free encyclopedia, "Java Native Interface," http://en.wikipedia.org/wiki/Java_Native_Interface, 2011, JNI is a programming framework that allows Java code to call native applications. 3
[5] D. Stefan, F. D. Cesare, A. Andrasescu, E. Popa, A. Lazariev, E. Vescovo, O. Strbak, S. Williams, Z. Starcuk, M. Cabanas, D. van Ormondt, and D. Graveron-Demilly, "Quantitation of magnetic resonance spectroscopy signals: the jMRUI software package," *Meas. Sci. Technol.*, vol. 20, p. 104035 (9pp), 2009. 3
[6] M. Fowler and K. Scott, *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1997. 3
[7] Oracle, "`Class ProcessBuilder`," http://docs.oracle.com/javase/6/docs/api/java/lang/ProcessBuilder.html, 2012, This class is used to create operating system processes. 3
[8] Java Tips, "From Runtime.exec() to ProcessBuilder," http://www.java-tips.org/java-se-tips/java.util/from-runtime.exec-to-processbuilder.html, 2012, ProcessBuilder should make it easier to invoke a subprocess with a modified process environment. 3

[9] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang, “Safe Java Native Interface,” in *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006, pp. 97–106, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.7019. 4

[10] Y. K. Hooi and A. Oxley, “Java Application Fault Tolerance Towards Unsafe Native Code Invocation Using ProcessBuilder,” in *In: 2011 3rd International Conference on Software Technology and Engineering*, Kuala Lumpur, August 12-13 2011, http://eprints.utp.edu.my/6397/. 4, 6

[11] J.W. van der Veen, R. de Beer and D. van Ormondt, “Increasing the computational speed of command line GammaPress. `PPSS`-controlled distribution of parallel processing on multi-core/hyper-threading enabled CPUs of multiple host computers,” Report on behalf of the Marie-Curie Research Training Network FAST, 2012, . 4

Appendix

Codes of the Java-based parallel-computing software system

A. The Java codes

1) `ParallelExecutor.java`: To start the Java application and submit the parallel tasks.

```
package concurrent.jni.ga;

import java.io.*;
import java.util.Date;
import java.util.concurrent.*;

public class ParallelExecutor {
  private int nr_tasks;
  private String[] pars;

  public ParallelExecutor(String[] args) {
    nr_tasks = Integer.parseInt(args[args.length - 1]);
    pars = new String[args.length];

    for(int i = 0; i < args.length; i++) {
      pars[i] = args[i];
    }
  }

  public void run() {
    System.out.println("");
    System.out.println("Hello in run() method of ParallelExecutor!");
    long time_begin = new Date().getTime();
    System.out.println("Time_begin (milli-seconds) = " + time_begin);
    System.out.println("");

    int nr_avail_proc = Runtime.getRuntime().availableProcessors();

    System.out.println("Number of available processors = " + nr_avail_proc);
    System.out.println("");

    ExecutorService execserv = Executors.newCachedThreadPool();
    CompletionService compserv = new ExecutorCompletionService (execserv);

    for(int i_task = 0; i_task < nr_tasks; i_task++) {
      compserv.submit(new ParallelTask(i_task, pars));
      try {
        Thread.sleep(100);
      }
      catch (InterruptedException e) {}
    }

    Object taskReturn;

    for(int i_task = 0 ; i_task < nr_tasks; i_task++) {
      try {
        taskReturn = compserv.take().get();
        System.out.println("Return of task(" + i_task + ") (seconds) = " + taskReturn);
      }
      catch (InterruptedException e) {}
      catch (ExecutionException e) {}
    }

    execserv.shutdown();

    long time_end = new Date().getTime();
    System.out.println("");
    System.out.println("Time_end (milli-seconds) = " + time_end);
    Double comp_time = new Double((time_end - time_begin)*0.001);
    System.out.println("Total computational time (seconds) = " + comp_time);
    System.out.println("");
  }

  public static void main(String args[]) {
    new ParallelExecutor(args).run();
    System.exit(0);
  }
}
```

2) `ParallelTask.java`: To carry out the parallel task by calling the `ProcessBuilder.start()` method.

```
package concurrent.jni.ga;

import java.io.*;
```

```java
import java.util.concurrent.*;
import java.lang.*;

public class ParallelTask implements Callable {
  private int index;
  private String[] pars;
  private ProcessBuilder procbuilder;

  public ParallelTask(int i_task, String[] args) {
    index = i_task;
    pars = new String[args.length];

    int lenminone = args.length - 1;
    for(int i = 0; i < lenminone; i++) {
      pars[i] = args[i];
    }
    pars[lenminone] = Integer.toString(index);
  }

  public Object call() {
    long begTest = new java.util.Date().getTime();
    System.out.println("Start of task(" + index + "):");

    try {
      String[] command = new String[4];
      command[0] = "java";
      command[1] = "-Djava.library.path=lib";
      command[2] = "concurrent/jni/ga/NativeC";
      command[3] = String.valueOf(index);

      procbuilder = new ProcessBuilder(command);
      Process proc = procbuilder.start();
      writeProcessOutput(proc);
    }
    catch (Exception e) {}

    Double secs = new Double((new java.util.Date().getTime() - begTest)*0.001);
    return secs;
  }

  void writeProcessOutput(Process process) throws Exception{
    InputStreamReader tempReader = new InputStreamReader(
      new BufferedInputStream(process.getInputStream()));
    BufferedReader reader = new BufferedReader(tempReader);
    while (true){
      String line = reader.readLine();
      if (line == null)
        break;
      System.out.println(line);
    }
  }
}
```

3) `NativeC.java`: To load the native library `libnativeC.so` and call the native C function `callNativeC()`.

```java
package concurrent.jni.ga;

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class NativeC {

  private static final long serialVersionUID = 1L;

  private String[] pars;

  public NativeC(String[] args) {
    String userdir_cur = System.getProperty("user.dir");
    //System.out.println("Current user.dir = " + userdir_cur);

    pars = new String[args.length];

    for(int i = 0; i < args.length; i++) {
      pars[i] = args[i];
    }
  }

  public void run() {
    //testNativeC();

    double[] dreturn = callNativeC(pars);

    for(int i = 0; i < dreturn.length; i++) {
```

```
      System.out.println("dreturn[" + i + "] = " + dreturn[i]);
    }
  }

  public static void main(String args[]) {
    new NativeC(args).run();
    System.exit(0);
  }

  public void testNativeC(){
    System.out.println("Hello Java world from NativeC!");
  }

  public native double[] callNativeC(String[] args);

  static {
    System.loadLibrary("nativeC");
  }
}
```

B. The C code

1) `callNativeC.c`: To call the Fortran subroutine `callfortran()` and return results to Java.

```
#include <jni.h>
#include "concurrent_jni_ga_NativeC.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

JNIEXPORT jdoubleArray JNICALL
Java_concurrent_jni_ga_NativeC_callNativeC(
  JNIEnv *env,
  jobject obj,
  jobjectArray args_java) {

  jstring tmp_string;
  tmp_string = (jstring) (*env)->GetObjectArrayElement(env, args_java, 0);
  int index  = atoi((*env)->GetStringUTFChars(env, tmp_string, 0));

  int i, len, i_return_len;

  double return_array[100];
  for (i=0; i<100; i++) {
    return_array[i] = 0.0;
  }

  callfortran_(&index, return_array, &i_return_len);

  len = (int) i_return_len;
  double dreturn[len];
  for (i=0; i<len; i++) {
    dreturn[i] = return_array[i];
  }

  jsize start = 0;
  jsize size = len;
  jdoubleArray jdreturn = (*env)->NewDoubleArray(env, size);
  (*env)->SetDoubleArrayRegion(env, jdreturn, start, size, (jdouble*) dreturn);

  return jdreturn;
}
```

C. The Fortran code

1) `callfortran.f`: To call the Fortran 90 subroutine `semipar()`. The code of the latter is not included. This subroutine is not the same as the *test* subroutine `semipar()` shown in Figure 4.

```
  subroutine callfortran(index, return_array, i_return_len)

  integer index, i, i_return_len
  double precision return_array(100)

  do i = 1, 100
    return_array(i) = 0.0d0
  enddo

  call semipar(index, return_array, i_return_len)

  return
  end
```

D. Miscellaneous

1) `compile.sh`: To compile the software system on a Linux Ubuntu 9.10 computer.

```
#!/bin/bash

PATH_PAREXE="/home/beer/Documents/parexe_ga_less_writes"
PATH_JAVA="/usr/lib/jvm/java-6-sun-1.6.0.24"

echo "PATH_PAREXE =" "${PATH_PAREXE}"
echo "PATH_JAVA =" "${PATH_JAVA}"

${PATH_JAVA}/bin/javac -Xlint ParallelExecutor.java ParallelTask.java NativeC.java

${PATH_JAVA}/bin/javah -jni -classpath ".:${PATH_PAREXE}" concurrent.jni.ga.NativeC

gfortran-4.4 -c -w callfortran.f pikaia.f90 semipar.f90

gcc -fPIC -I "${PATH_JAVA}/include" -I "${PATH_JAVA}/include/linux" -shared -lgfortran
    -lm -o libnativeC.so callNativeC.c callfortran.o pikaia.o semipar.o

cp libnativeC.so "${PATH_PAREXE}/lib"
```

2) `run.sh`: To run the software system on a Linux Ubuntu 9.10 computer.

```
#!/bin/bash

PATH_JAVA="/usr/lib/jvm/java-6-sun-1.6.0.24"
${PATH_JAVA}/bin/java -Djava.library.path=lib concurrent/jni/ga/ParallelExecutor 2
```