ELSEVIER

Future Generation Computer Systems 17 (2001) 987–997

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.nl/locate/future

Convenient use of legacy software in Java with Janet package

Marian Bubak [a,b,*], Dawid Kurzyniec [c,1], Piotr Łuszczek [d,2]

[a] *Institute of Computer Science, AGH al. Mickiewicza 30, 30-059 Kraków, Poland*
[b] *Academic Computer Centre — CYFRONET, Nawojki 11, 30-950 Kraków, Poland*
[c] *Department of Mathematics and Computer Science, Emory University, 1784 North Decatur Road, Atlanta, GA 30322, USA*
[d] *Department of Computer Science, University of Tennessee, 1122 Volunteer Blvd., Suite 203, Knoxville, TN 37996-3450, USA*

Abstract

This paper describes *Janet* package — highly expressive Java language extension that enables convenient creation of powerful native methods and efficient Java-to-native code interfaces. Java native interface (JNI) is a low-level API that is rather inconvenient if used directly. Therefore Janet, as the higher-level tool, combines flexibility of JNI with Java's ease-of-use. Performance results of Janet-generated interface to the `lip` library are shown. Java code, which uses `lip`, is compared with native C implementation.

Keywords: High-performance Java; Java native interface; Out-of-core computing

1. Introduction

Rapid evolution of Java [10] made this language suitable for high-performance computing [1,11,12,16,18,21]. It may be attributed mostly to the fact that Java is portable (i.e., suitable for heterogeneous environments), simple (i.e., easy to learn), safe (i.e., facilitates debugging), secure (i.e., enables secure distributed computing), and modern (i.e., is object oriented, employs automatic memory management and uses exceptions to handle erroneous situations).

With the performance of Java virtual machines (VMs) continuously increasing [12,18], the main issue becomes the lack of scientific libraries designed for use in Java [7]. This may be overcome with the use of Java native interface (JNI) [13,20] which makes it possible to integrate Java with existing C/C++ and Fortran code. In practice, such integration proved to be complicated and so several tools which automate interface creation process have been developed [8,17]. Automation, however, came at the cost of making the use of the JNI interfaces very cumbersome and error-prone since at all times the user is required to work at the level of a native machine rather than virtual one. The *Janet* (JAva Native ExTensions) [3] package which we describe in this article automates the creation process of JNI interfaces and still gives full access to all Java features so that the resulting code may refer to Java variables, objects and classes, throw and handle Java exceptions, synchronize on Java monitors, access Java strings and arrays.

* Corresponding author. Tel.: +48-12-617-39-64; fax: +48-12-633-80-54.
E-mail addresses: bubak@agh.edu.pl (M. Bubak), dawidk@mathcs.emory.edu (D. Kurzyniec), luszczek@cs.utk.edu (P. Łuszczek).
[1] Tel.: +1-404-412-8665.
[2] Tel.: +1-865-974-8295; fax: +1-865-974-8296.
[3] Home page of the Janet project: http://www.icsr.agh.edu.pl/janet/

0167-739X/01/$ – see front matter
PII: S0167-739X(01)00041-3

This functionality is available through ordinary Java syntax instead of inconvenient and complicated JNI function invocations.

2. Overview of JNI

The JNI [13,20] allows Java code running inside a Java VM to interoperate with applications and libraries written in other programming languages, such as C/C++ or Fortran. Java methods, which were declared `native` in Java class definition, are implemented in a native language. The native code may use JNI calls to interoperate with the running JVM. Design of JNI enables the native method to perform all the operations that are allowed for a Java code. However, this functionality is available through a complex set of routines and their parameters and its abstraction is rather low which makes the development process long, inconvenient and error-prone. Most of the programming mistakes in created interfaces (which are rather easy to make) lead to runtime errors, which are hard to track not only because they may appear only under certain circumstances but also are platform-specific (as are the native languages supported by JNI). Some of such error-prone situations include the following are the examples of:

Access to Java arrays. To reference Java arrays, native code must invoke special JNI function to lock the array and obtain the pointer to it. When the array is no longer needed, another function must be called to release the array. JNI specification does not define the behavior of a program which fails to release the array. In addition, distinct functions must be used to deal with arrays of different types. The use of improper functions causes runtime errors rather than compile-time errors. This issue is also related to the access to Java strings.

Access to fields of Java objects. It requires field descriptors which must be obtained first by using: reference to the class, field name, and its type *signature* [13]. There are separate lookup functions for static and instance fields. Also, the accessor functions are different for fields of different types.

Invoking Java methods. It is difficult because method signature includes type signatures of its parameters. There are also different JNI functions for different invocation modes: instance, static, and non-virtual. As a result, when the field type or declared parameter type of a method changes, the interface code becomes invalid and it has to be modified (in contrast, Java code which uses affected field or method only has to be recompiled).

Handling exceptions. There is no Java-style way to handle exceptions, which may occur in native methods as a result of JNI calls (e.g., when Java methods are invoked). Checking whether exception has been thrown requires explicit query, which is mandatory as the behavior of subsequent JNI calls is undefined when there are pending exceptions.

Working with Java monitors. Locking and unlocking of Java monitors are two independent operations so frequently the latter is mistakenly omitted at the runtime, especially within exception handling code.

3. Janet overview

Janet package enables convenient development of native methods and Java-to-native code interfaces by completely hiding the JNI layer from the user. Still, functionality that can be achieved using JNI may be also achieved with Janet using ordinary Java syntax.

Janet source file is similar to ordinary Java source file except that it may contain embedded native code. This is achieved with back-tick (‘) characters as native code delimiters. The native code can easily and directly access Java variables as Java code would (no need for JNI calls which are inserted by Janet translator).

Janet file is transformed by Janet translator into Java and native language source files, as shown in Fig. 1. The code that is automatically generated for the user performs, among others, the following operations: determines necessary type signatures, chooses JNI functions to call, load Java classes, obtains field and method descriptors,

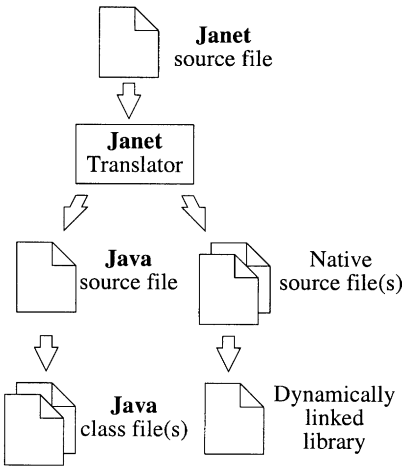

Fig. 1. Janet translation process.

File HelloWorld.janet:

```
class HelloWorld {
native "C" {
#include <stdio.h>
}
  public native "C" void displayHelloWorld() {
    printf("Hello world!\n");
  }
  public static void main(String[] args) {
    new HelloWorld().displayHelloWorld();
  }
}
```

File HelloWorld.java:

```
class HelloWorld {
  /* ... code that loads library goes here ... */
  public native void displayHelloWorld();
  public static void main(String args) {
    new HelloWorld().displayHelloWorld();
  }
}
```

File HelloWorldImpl.c:

```
#include <janet.h>
#include <stdio.h>
Janet_HelloWorld_displayHelloWorld(JNIEnv *_janet_jnienv,
                                   jobject _janet_obj)
{
  printf("Hello world!\n");
}
```

Fig. 2. “Hello World” program in Janet and resulting Java and C files.

performs array and string lock and release operations, handles and propagates Java exceptions, and matches monitor operations.

The simple example of canonical "Hello World" program using Janet is presented in Fig. 2, together with the files resulting from the translation process.

3.1. Embedding Java expressions into native code

Perhaps the most important feature of Janet is that it allows to embed into native code Java expressions and statements such as class instance creations, string and array operations, field accesses, method invocations, comparisons, runtime type checks involving use of `instanceof` operator, assignments, and more.

Below, the example from the interface to the `lip` [6] runtime library is shown where native function is invoked on the pointer fetched from the Java object passed as a native method argument.

```
public static native void maptableFree(Maptable mtab) {
  LIP_Maptable_free(‘mtab.data‘);
}
```

Janet enables straightforward use of Java arrays at the native side. It also allows to use native code inside embedded Java expressions, which is useful when indexing Java arrays with native variables:

```
native int sum(int[] arr) {
  int len=‘arr.length‘; /* embedded Java expression */
  int i, sum=0;
  for (i=0; i<len; i++) {
    sum+=‘arr[#(i)]‘; /* C variable ‘i’ inside Java code */
  }
  return sum;
}
```

The previous example also shows a feature of Janet called recursive embedding. First, the method signature conforms to the Java native method syntax. Second, the native method body constitutes of C code with Java code embedded in back-ticks (‘). This embedded Java may contain C expressions with *#*(*expr*) syntax which in turn may contain another level of embedded Java (again within back-ticks).

Janet guarantees that the evaluation order of embedded Java expressions is exactly the same as specified in ([10], Sections 15.5 and 15.6). It means, e.g., that method arguments are always evaluated from left to right, and, what is more important, Java exceptions immediately break the evaluation process in the place where they occurred.

Java variables can be declared inside a native method implementation and used in subsequent embedded Java expressions:

```
native void method(BookStore bs) {
  ‘Book b;‘
  // ...
  ‘b=bs.getBook();‘
  // ...
  printf(’’%d\n’’,(int)‘b.getPageCount()‘);
}
```

Use of such variables to hold array references makes it possible to gain control over occurrence of the array lock/release operations. Locked array will not be released as long as there are some variables referencing it. Without such variables, the array is released upon reaching the end of the block containing array access expression, or when the expression is re-evaluated producing different array reference. This is shown by the following code:

```
class Dummy {
  int[] arr0;
  int[] arr1;
  native void method() {
    ‘int local[];‘
    int val1, val2;
    {
      val1=‘arr0[0]‘;                          /* lock of arr0 */
      ‘local=arr1;‘                            /* new reference to arr1 */
      val2=‘local[0]‘;                         /* lock of arr1 */
    }                                          /* arr0 released */
  }                                            /* arr1 released */
}
```

3.2. Exception handling

Exception handling is one of the most error prone aspects of JNI. The user must explicitly check for exceptions in every possible place where they may occur, i.e., after most of JNI calls, and provide code for handling them. As exceptions should usually break normal flow of program execution, it becomes easy to decouple array or monitor lock/release operation pairs in the exception handling code.

Janet provides safe Java-style exception handling model at the native side. This problem is addressed by allowing use of Java `try`, `catch`, `finally` and `throw` statements inside the native code. It guarantees that all Java arrays and monitors are released no matter if exception was thrown or not. Exceptions are always handled by the nearest applicable `catch` clause. Uncaught exceptions immediately stop native method execution and propagate to the Java side as does the `IllegalArgumentException` in the example below:

```
native void method() {
  ‘try‘{
     ‘callback();‘
  }‘catch (Throwable e)‘{
    JNI_EXCEPTION_DESCRIBE();
    ‘throw new IllegalArgumentException(‘‘thrown from C code’’);‘
  }
}
```

In this example, native code calls Java method `callback()` which can cause an exception. `JNI_EXCEPTION_DESCRIBE()` is the C macro which uses JNI function that prints the exception stack.

Using pure JNI, the method body of this simple example requires 18 lines of code instead of six (see [20]).

3.3. Synchronization

Rather than providing separate functions for monitor lock and unlock operations as JNI does, Janet allows to use Java `synchronized` statement inside native code:

```
native void foo(Object bar) {
  ‘synchronized(bar)‘{
    native_foo();
  }
}
```

In this way monitors are guaranteed to be eventually unlocked even if the exception occurs inside `synchronized` body.

3.4. Arrays

As it has been shown Janet enables efficient direct access to Java arrays from the native side. However, when the array is to be processed by external routine the array pointer has to be used. To obtain the pointer, Janet provides the address fetch operator (&) applicable to array references:

```
native void qsort(int[] arr) {
  jint* ptr;
  ptr=‘& arr‘;
  qsort(ptr, ‘arr.length‘, sizeof(jint), ...);
}
```

Java arrays contain platform-independent Java primitive data types rather than native ones, and these types are not necessarily the same (`jint`, `jlong`, `jboolean`, `jchar`, `jbyte`, `jshort`, `jfloat` and `jdouble` are native equivalents of Java primitive data types as defined by the JNI). When the array of native data type is desired the special #& operator may be used on an array reference to perform conversion.

```
native void polint(float[] xa, float[] ya, ...) {
  polint(‘#&xa‘, ‘#&ya‘, ...);
}
```

The general rule is that Java types are converted by Janet to native types with the corresponding names. The array conversion introduces no performance reduction on platforms where appropriate Java and native types are equivalent, but it requires allocation and copying of the whole array in the case when they are different.

3.5. Unstructural flow of execution

The usual programming practice inherited from C can lead to a native method code like this:

```
do {
  ‘synchronized(foo)‘ {
    break;
  } /* monitor unlock would occur here */
} while (false);
```

This code leaves the associated Java monitor locked until the native method returns. C language (unlike, e.g., C++) is not expressive enough to enable Janet to handle such situations properly without performing sophisticated analysis of execution flow. For that reason, Janet forbids in C the use of unstructural flow statements, namely `break`, `continue`, `goto` as well as `longjmp()` function call, that would bail out of the block in which they appear (early `return` is allowed but it is going to become deprecated in future versions as it does not guarantee proper evaluation of possible `finally` clauses of surrounding `try` statements — the Java style `return` statement will be introduced instead).

3.6. Performance

Performance of every JNI-based Java-to-native code interface strongly depends on the performance of JNI implementation used in given Java VM. Our tests have shown that substantial differences may exist even between VMs which otherwise perform similarly. Moreover, there is absolutely no guarantee that JNI performance is going to

improve in subsequent versions of any given Java VM (even from the same vendor), in fact, it may as well decrease as more sophisticated garbage collection techniques and JIT optimizing algorithms are used. This was the case for both HotSpot Client and Server VMs from SUN's Java 2 standard edition 1.3 for Linux, which perform JNI calls much slower than older and generally less efficient Classic VM.

Two most important performance issues that users should be aware of, consider the access to arrays of primitive data types and invocation of callback methods. According to the specification [13], Java VM is not required to avoid copying of the whole array (instead of "pinning it down") when the array pointer is requested through JNI call. Although all of the JIT-enabled VMs which we have tested were able to pin down the arrays accessed through `GetArrayCritical()` routine [13], only few of them (SUN HotSpot Client for Win32, IBM VM 1.3.0 for Win32, and IBM VM 1.3.0 for Linux) achieved it when less restrictive `Get⟨type⟩Array()` routines were used so the routines of latter type should rather be avoided if possible, as they can degrade the performance when arrays

```
class Test extends LipNode {
  ...
  protected int startComputing() throws LipLibraryException {
    File cf = new File(); // create lip file object
    double[] x, y;
    int[] indices, l_indices; // indirection arrays:
                              // global and local

    // Maptable describes data distribution
    mtab = new Maptable(Lip.MAP_BLOCK, ...);

    // open file with index array
    cf.open( "indices", MPI.MODE_RDONLY ...);

    for (/* all i-sections */) {
      cf.read(...); // read i-section data into local array

      // Index translation and communication
      // schedule generation.
      schedule = Lip.localize(mtab, indices, l_indices, ...);
      Lip.gather(..., schedule); // gather data
                                 // from other nodes

      for (j=0; j<is_size; j++) {  // perform computation
        y[l_indices[j]] += f( x[l_indices[j]], n);
      }
      Lip.scatter(..., schedule, MPI.SUM); // scatter computed
                                           // data back
    }
    cf.close();
    ...
  }
}
```

Fig. 3. Sample Java code for OOC computing with the `lip`.

become large enough. Callback method invocation overheads tended to vary between different VMs up to an order of magnitude, especially for methods with large number of arguments, and were about one order of magnitude more expensive than the rest of JNI functions. For this reason, intensive callback invocations should be performed with caution and tested on different VMs.

3.7. Portability

One of the main goals of Janet project is to retain high level of portability of both the tool and resulting code. The Janet translator is thus written entirely in Java and it can run on Java 2 platform version 1.2.2 and higher. The generated native code fully conforms to the ANSI C standard and can cooperate with JNI starting from version 1.1 so it will work with JDK 1.1, but it can also benefit from the JNI 1.2 extensions on newer platforms.

The *Janet* project has grown from the Java interface to the `lip` runtime library [6,3,4]. The `lip` library is built on top of the MPI [15,9,14] and supports both in- and out-of-core (OOC) parallel irregular problems [19,2]. The `lip` is built on top of the Message Passing Interface (MPI). The use of the MPI [15] as a communication layer makes the `lip` portable. At the same time, there is no support for solving irregular and out-of-core (OOC) problems in Java, whereas such a support is needed [21]. The latter, together with the portability of the `lip`, were the most compelling reasons to choose this library as an example usage of Java-to-native interface created using the techniques described above. The tests of generated Java interface to `lip` were performed [5,3] and they proven that Java can be efficiently employed to such large scale scientific parallel computations. It introduces rapid software development and safety to existing native communication layer.

4. Experimental results

Fig. 3 presents a sample of Java version of an OOC test program. The code demonstrates a generic irregular OOC problem. There are two *data arrays* — `x` and `y`, together with *indirection arrays*: `indices` and its local version `l_indices`. The indirection arrays may be so large that they cannot fit into main memory and, therefore, must be stored on disk. The data arrays are distributed among the computing nodes. During the computation phase, transformations of data arrays are performed. Since the data arrays are indexed through indirection arrays which are altered during the program execution, the data access pattern is not known until runtime.

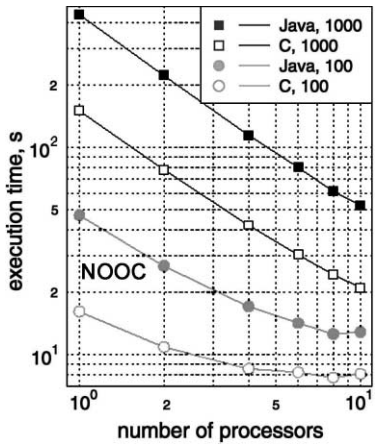

Fig. 4. Execution time for OOC problem in Java and C, $n = 100$ and $n = 1000$.

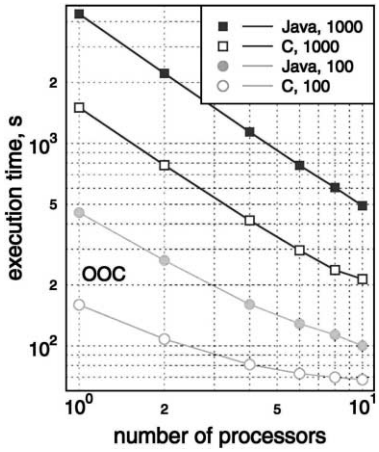

Fig. 5. Execution time for OOC problem in Java and C, $n = 100$ and $n = 1000$.

The full test programs in C and Java from Fig. 3 were run on a cluster of 10 Linux PCs each with 333 MHz i686 Celeron processor and 32 MB physical memory. The machines were connected with ethernet and each PC was equipped with 2 GB local hard disk, while approximately 600 MB of it being available for the program use. The LAM6.3b1 [14] was used as an MPI implementation together with the Java VM from Java Development Kit version 1.1.7 for Linux. The workload incurred by the call to `f()` (see Fig. 3) function was controlled with parameter n. Each of the tests was run twice in C and in Java for two different values of n. The data arrays were of type `double[]`, whereas index arrays were of type `int[]`.

Fig. 4 shows a small test case where both data and index arrays are small enough to fit into the main memory (there are three arrays of 360 360 values each). Fig. 5 shows an OOC case where the index array is large enough (3 603 600 `int` values) so that it does not fit into the main memory and therefore must be stored on disk. Fig. 6 presents timings for a *constant-time problem* where both data and index array sizes were proportional to the total number of the computing nodes N (the array sizes were $180\,180 \times N$ values).

It is worth noting that the scalability of the problem in C and Java is very similar, and it depends on the *computation-to-communication* ratio. For $n = 100$, there is a noticeable saddle where the communication and

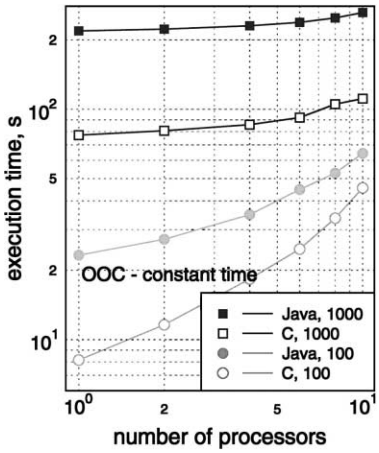

Fig. 6. Execution time for OOC problem in Java and C, constant time problem, $n = 100$ and $n = 1000$.

I/O costs excess those of computations. It is also important that our problem is a generic one and has random data distribution, thus it requires a large amount of communication whenever the data is assigned to the computing nodes.

5. Conclusions and future work

This paper describes the new approach to creation of Java-to-native code interfaces. The presented Java language extension called Janet enables simple, fast and efficient development of such interfaces retaining full control over their low-level behavior. At this point, our goal is to provide a higher-level tool (with graphical-user interface) thus enabling the user to graphically design the structure of Java wrappers for a native library. As a result, the tool would generate Janet code which could be further refined by the user. The fully automatic wrapper generator is also under consideration with its output being subject to potential refinement by the GUI tool. At the same time, we intend to apply Janet to enable usage of native resources in the Harness metacomputing framework [11,16].

Acknowledgements

This research was done in the framework of the Polish–Austrian collaboration and it was supported in part by the KBN Grant 8 T11C 006 15. Dawid Kurzyniec extends special thanks to Prof. Vaidy Sunderam from Department of Mathematics and Computer Science at Emory University in Atlanta.

References

[1] R.F. Boisvert, J.J. Dongarra, R. Pozo, K.A. Remington, G.W. Stewart, Developing numerical libraries in Java, in: Proceedings of the ACM-1998 Workshop on Java for High-performance Network Computing, Stanford University, Palo Alto, CA. http://www.cs.ucsb.edu/conferences/java98/papers/jnt.ps.

[2] P. Brezany, Input/output intensively parallel computing, Lecture Notes in Computer Science 1220, Springer, Berlin, 1997.

[3] M. Bubak, D. Kurzyniec, P. Łuszczek, A versatile support for binding native code to Java, in: M. Bubak, H. Afsarmanesh, R. Williams, B. Hertzberger (Eds.), Proceedings of the HPCN Conference, Amsterdam, May 2000, pp. 373–384. ISBN 3-540-67553-1.

[4] M. Bubak, D. Kurzyniec, P. Łuszczek, Creating Java-to-native code interfaces with Janet extension, in: M. Bubak, J. Mościnski, M. Noga (Eds.), Proceedings of the First Worldwide SGI Users' Conference, ACC-CYFRONET, Cracow, October 11–14, 2000, pp. 283–294. ISBN 83-902363-9-7.

[5] M. Bubak, D. Kurzyniec, P. Łuszczek, Out-of-core computing in Java, in: T. Szmuc (Ed.), Proceedings of the Cracow Centre for Advanced Training in Information Engineering Conference, AGH, Cracow, October 1999.

[6] M. Bubak, P. Łuszczek, Towards portable runtime support for irregular and out-of-core computations, in: J. Dongarra, E. Luque, T. Margalef (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the Sixth European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999, Springer, Berlin, Lecture Notes in Computer Science, pp. 59–66.

[7] E.D. Demaine, Converting C pointers to Java references, in: Proceedings of the ACM-1998 Workshop on Java for High-performance Network Computing, Stanford University, Palo Alto, CA. http://www.cs.ucsb.edu/conferences/java98/papers/pointers.ps.

[8] V. Getov, S. Flynn-Hummel, S. Mintchev, High-performance parallel programming in Java: exploiting native libraries, in: Proceedings of ACM-1998 Workshop on Java for High-performance Network Computing, Stanford University, Palo Alto, CA, 1998. http://www.cs.ucsb.edu/conferences/java98/papers/hpjavampi.ps.

[9] V. Getov, P. Gray, V. Sunderam, MPI and Java-MPI: contrasts and comparisons of low-level communication performance, in: Proceedings of SuperComputing '99, Portland, OR, November 13–19, 1999.

[10] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, Reading, MA, 1996.

[11] Harness project home page. http://www.mathcs.emory.edu/harness.

[12] Java Grande forum. http://www.javagrande.org/.

[13] JavaSoft. Java native interface. http://java.sun.com/products/jdk/1.2/docs/guide/jni/.

[14] LAM/MPI parallel computing. http://www.mpi.nd.edu/lam/.

[15] Message passing interface forum, MPI-2: extensions to the message-passing interface, July 18, 1997. http://www.mpi-forum.org/docs/mpi-20.ps.

[16] M. Migliardi, V. Sunderam, The harness metacomputing framework, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, South Antonio, TX, March 22–24, 1999. http://www.mathcs.emory.edu/harness/PAPERS/pp99.ps.gz.
[17] S. Mintchev, V. Getov, Towards portable message passing in Java: binding MPI, in: M. Bubak, J. Dongarra, J. Waśniewski (Eds.), Proceedings of the Fourth European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Cracow, Poland, November 1997, Springer, Berlin, Lecture Notes in Computer Science 1332, 1997, pp. 135–142.
[18] M. Philippsen, Is Java ready for computational science? in: Proceedings of the Second European Parallel and Distributed Systems Conference, Vienna, July 1998. http://math.nist.gov/javanumerics/.
[19] J. Saltz, et al., A Manual for the CHAOS Runtime Library, UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, March 1995. ftp://ftp.cs.umd.edu/pub/hpsl/chaos_distribution/.
[20] B. Stearns, Trail: Java native interface. http://ftp.javasoft.com/docs/tutorial.zip, directory native1.1/.
[21] G. Zhang, B. Carpenter, G. Fox, X. Li, Y. Wen, The HPspmd model and its Java binding, in: R. Buyya (Ed.), High-Performance Cluster Computing, Vol. 2: Programming and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1999 (Chapter 14).

Marian Bubak received his MSc degree in Technical Physics and PhD in Computer Science from the University of Mining and Metallurgy (AGH), Kraków, in 1975 and 1986, respectively. Since graduation he worked at the Institute of Physics and Nuclear Techniques AGH, and in 1982 he moved to the Institute of Computer Science AGH. At present he is an Assistant Professor (adjunct) at the Institute of Computer Science AGH and at the Academic Computer Center CYFRONET AGH, Kraków, Poland. His current research interests include parallel computing and tool support for distributed applications and systems. He has been served as the program committee member and organizer of several international conferences in the area of Computational and Computer Science.

Dawid Kurzyniec was born in Kraków, Poland. He received his MSc degree in Computer Science at the University of Mining and Metallurgy in Kraków in 2000. Currently, he is working as a research associate in the Department of Math and Computer Science at Emory University in Atlanta. His research interests include distributed computing and object oriented technologies.

Piotr Łuszczek was born in Kraków, Poland and received his MSc degree in Computer Science at the University of Mining and Metallurgy in Kraków in 1999. He is currently a PhD student in Computer Science Department at the University of Tennessee, Knoxville, TN. His research interests include distributed computing and sparse linear algebra.