Abstract. Now Java is considered as an appropriate environment for high performance computing. In this paper, we describe the Java wrappers for the lip—a runtime system which enables easy and portable parallelization of irregular and out-of-core computations. The sample results of Java out-of-core computations are also presented.

1 Introduction

Recently, there is a growing interest in application of the Java language in the high performance computing (see e.g., [1, 2, 3].) It is mostly due to Java’s doubtless advantages: portability, ease of use and learn, safety features, and availability on the wide range of platforms. At the same time, many people try to cope with Java’s weakness, i.e. poor performance and limited expressiveness, so there is a great chance that soon Java will outperform C and Fortran [2].

Currently, there are many scientific libraries which consist of hundreds of thousands lines of code in Fortran, C or other languages. These libraries are tuned for numerous platforms, they are thoroughly tested and well known. Hence, there is a strong demand for them and the most appropriate way to make them available in Java is to provide pertinent interface for them.

lip [4] is a portable runtime support library for both in- and out-of-core irregular problems. Irregular problems are parallelized in the similar way as in the CHAOS library [5] whereas the out-of-core functions are based on the idea of in-core section [6]. There is no support for solving irregular and out-of-core (OOC) problems in Java, while there is a need for such a support [1]. At the same time, the lip library supports such problems in C and Fortran. The lip is built on top of the Message Passing Interface (MPI). The use of the MPI [7] as a communication layer makes the lip portable. This portability together with the ability to support irregular OOC problems were the reasons why we have decided to develop the Java wrappers to the lip library.

2 Design Goals

The JCI tool is the automatic Java-to-C interface generator [8, 9]. With this tool, it is possible to create such an interface with a very little effort. However, JCI does not allow to use Java-specific features, such as: garbage collector for automatic memory deallocation, object oriented programming, exceptions etc. Using JCI also makes the Java programs unsafe: C libraries typically do not check the correctness of function arguments. A call to such a function from Java with a wrong value of an argument will crash the Java Virtual Machine (JVM). In order to avoid this, a wrapper should check the arguments before calling actual function. Many of these issues can be fixed by a manual refinement of the wrappers after their generation; however, it often requires a lot of additional work and results in lower performance.

Due to the reasons mentioned above, a different approach was taken for the development of Java wrappers for the lip library. The lip’s functionality is expressed by a careful choice of arguments rather than numerous set of functions. Therefore, semi-automatic way of generating wrappers seems not only feasible but also advantageous. It allows the use of the following Java-specific features:

- **Object oriented paradigm.** The lip handles, such as LIP Schedule, are mapped to Java classes; the library functions which are logically connected with the handles are mapped to class methods and constructors of those classes.

- **Automatic memory deallocation.** In C, lip objects like LIP Schedule should be freed by calling appropriate LIP_\*free() functions.
Figure 1: Sample C code for OOC computing with lip.

```c
#include "lip.h"
int main(int argc, char **argv) {
    LIP_file cf;
    ...
    double sum, buf;
    LIP_Schedule schedule;
    LIP_Maptable mtab;
    ...
    double *x, *y;
    int *indices, *l_indices;
    ...
    MPI_Init( &argc, &argv );
    LIP_Setup( 0, 0 );
    LIP_RANK( &any_rank );
    /* fetch parameters from argv */
    /* Allocate memory for x, y and indices */
    /* Fill data array - local data array part */
    /* Fill indices array */
    LIP_file_open( "indices", MPI_MODE_CREATE |
                   MPI_MODE_WRONLY, MPI_INT, 0, LIP_TYPE_LOCAL,
                   NULL, &cf );
    for (/* all i-sections*/) {
        for (j = 0; j < is_size; j++) indices[j] = ...;
        LIP_file_write( cf, indices, is_size );
    }
    LIP_file_close ( &cf );
    LIP_Maptable_create(&mtab, LIP_MAP_BLOCK, &gdsize, 1);
    LIP_file_open( "indices", MPI_MODE_RDONLY ... , &cf);
    for (/* all i-sections */) {
        cf.read(J2N.section(indices, 0, is_size), 1.0f);
        /* index translation and schedule generation */
        schedule = Lip.localize(mtab,
                                 J2N.section(indices, 0, is_size),
                                 J2N.section(l_indices, 0, is_size),
                                 data_size, ghost_area_size);
        /* import data */
        Lip.gather(J2N.section(x),
                   J2N.section(x, data_size), schedule);
        /* compute */
        for (j=0; j<is_size; j++)
            y[l_indices[j]] += f( x[l_indices[j]], n );
    }
    /* export data */
    LIP_Scatter(J2N.section(y, data_size, y, schedule, MPI_DOUBLE, MPI_SUM);
    }.
    LIP_file_close( &cf );
    /* find total */
    sum = 0.0;
    for (i = 0; i < data_size; i++) sum += y[i];
    MPI_Reduce( &sum, &buf, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (my_rank) printf("Total: %g\n", buf);
    LIP_Finalize();
    return 0;
}
```

Figure 2: Sample Java code for OOC computing with the lip.

```java
import pl.edu.agh.icsr.lion.lip.*;
import pl.edu.agh.icsr.lion.mpi.*;
...
class Test extends LipNode {
    ...
    protected int startComputing(String[] args) throws LipLibraryException {
        File cf = new File(); /* lip file object */
        double sum, buf;
        Schedule schedule;
        Maptable mtab;
        ...
        double[] x, y;
        int[] indices, l_indices;
        /* fetch parameters from argv */
        /* Allocate memory for x, y and indices */
        /* Fill data array - local data array part */
        /* Fill indices array */
        cf.open("indices", Mpi.MODE_CREATE |
                   Mpi.MODE_WRONLY, Mpi.INT, 0,
                   LIP_TYPE_LOCAL, null);
        for (/* all i-sections */) {
            for (j=0; j<is_size; j++) indices[j] = ...;
            cf.write(J2N.section(indices, 0, is_size));
        }
        cf.close();
        int[] tmp_arr = { gdata_size };
        mtab = new Maptable(Lip.MAP_BLOCK, tmp_arr);
        cf.open("indices", Mpi.MODE_RDONLY ...);
        for (/* all i-sections */) {
            cf.read(J2N.section(indices, 0, is_size), 1.0f);
            /* index translation and schedule generation */
            schedule = Lip.localize(mtab,
                                     J2N.section(indices, 0, is_size),
                                     J2N.section(l_indices, 0, is_size),
                                     data_size, ghost_area_size);
            /* import data */
            Lip.gather(J2N.section(x),
                       J2N.section(x, data_size), schedule);
            /* compute */
            for (j=0; j<is_size; j++)
                y[l_indices[j]] += f( x[l_indices[j]], n );
        }
        /* export data */
        Lip.scatter(J2N.section(y, data_size, y, schedule, Mpi.SUM);
    }
    cf.close();
    /* find total */
    sum = 0.0;
    for (i = 0; i < data_size; i++) sum += y[i];
    ...
    Mpi.COMM_WORLD.reduce(J2N.section(vsum),
                          J2N.section(vbuf), 1, Mpi.DOUBLE, Mpi.SUM, 0);
    if (myNode == 0) System.out.println("Total: " + vbuf[0] +"\n");
    return 0;
}
```

when they are no longer used to reuse occupied memory (library does not free its objects until LIP_exit() is called). In Java version the LIP_free() functions are called automatically when objects are garbage collected. So, for example, user can safely assign new value to some lip reference; garbage collector will perform any required deallocations for native structures which were connected to this reference previously.

- **Exception raising.** In C, the lip functions return a value, and if this value is different from
LIP_SUCCESS this means error. Return-values are often ignored by programmers; using exceptions is a better solution since they may not be ignored as they immediately stop further program execution. So Java wrappers to the lip functions are declared so they could throw LipLibraryException.

In Figs 1 and 2 the sample OOC test programs in C and Java are shown. We used these programs in the test runs described in the Section 4. As one can see, both programs look quite similar except for employing Java features mentioned above on the one hand, and some unavoidable obstacles in Java program on the other (e.g., indirect access to array sections). As the lip is built on the top of the MPI, and lip applications are allowed to call the MPI routines directly, we provide skeleton Java wrappers of the MPI using the C++ MPI bindings specification presented in [7].

During the creation of the lip and the MPI wrappers, the wrapper-developer's library was elaborated which can be used with a little effort to create elegant wrappers to other native libraries.

3 Description of Implementation

Any programming library can be used in a program in a quite straightforward way by employing either static or dynamic linking mechanism as it is shown in Fig. 3.

During the creation of the lip and the MPI wrappers, the wrapper-developer's library was elaborated which can be used with a little effort to create elegant wrappers to other native libraries. tually an extra code which is accessible by JVM and calls functions of a native library.

The dynamic linkage of the wrapper code is performed by JVM with a call to System.loadLibrary(). The communication
between Java and the native code is done with functions specified by the Java Native Interface (JNI) [10]. The most important feature of JNI is that it enables the wrapper code to have the same functionality as pure Java, i.e., it is able to create objects, call methods, raise exceptions, perform computations on Java variables. The problem can arise while performing floating-point operations

For the use of the MPI. The library which implements the MPI is accessed by the MPI and it can also be accessed directly by the user (see Fig. 5). The wrappers for the MPI and MPI use common wrapper components at both Java and native side. These components (common wrapper classes and common wrapper library) contain a generic code, which can be used to wrap any C library. This code provides mechanisms for safe conversion of arguments between C and Java. There are numerous problems with such conversion (see [8, 9]), e.g.:

- primitive types in C have variable size which may differ from standard Java sizes on different platform,
- there is no direct analog of C pointers in Java [11],
- multidimensional arrays have no direct counterpart in Java,
- C structures can be emulated by Java objects, but the layout of fields of an object may differ between JVMs,
- C functions passed as arguments have no direct counterparts in Java.

A set of functions and macros was provided to resolve some of these issues, in particular, simplification of checking input parameters, throwing Java exceptions, dealing with arrays and array sections, etc. For example, only one macro ENSURE_NOT_NULL(obj, "objname") is needed to ensure that a reference to an object passed from Java to C has not the null value, and, if so, to throw a Java exception with an appropriate message, and to return immediately to the Java side.

Idea of employing C preprocessor macros to simplify wrappers creation has been introduced in JCI [8] as well as the idea of passing array sections from Java to C. Java does not have pointer arithmetic so passing “pointer to the middle of array” is impossible. We have extended that approach by introducing the idea of virtual arrays. Each object of such a type either encapsulates normal Java array directly or represents a subsection of another object of the same type. The objects can be declared with fixed or variable length. In the latter case, when passed to wrapper function, the length of the object can be changed (by reallocating underlying Java array) to satisfy the native function requirements before its invocation. In the example presented in Fig. 6, buf1 and buf2 are containers filled by a native function. buf1 is declared to have length of exactly 10 elements, while buf2 may be changed in the native library side as needed.

```
// we don't know the desired size, // so 20 is a starting point
Dvarr x = new Dvarr(new double[20]);

// exactly 10 first elements
Dvarr buf1 = x.section(0, 10);

// from 10th, no limit
Dvarr buf2 = x.section(10);

// fit in 10 or throw an exception
some_object.someMethod(buf1, ...)

// accept as much as needed
some_object.someMethod(buf2, ...)

double zb[i] = x.baseArray();
for (int i=10; i<xb.length; i++)
    doSomethingWith(xb[i]);
```

Figure 6: Virtual arrays.

We have also applied the idea of acquiring array sections by calling a static method with the array object, as shown in Fig. 2. This approach, introduced in JCI, is useful in large-scale computations, where arrays are large and allocated only once.

Another important feature of the wrappers, which allows employing Java garbage collector to manage native data structures, is an abstract Java class being able to encapsulate any C pointer or a handle – DataEncapsulator. Object of the class derived from DataEncapsulator holds the C pointer or the handle of some specific type, casted to a private field of type long. The object knows how to free the data structure which is associated with. It is done upon the request from garbage collector. This is shown in Fig. 2, where the Mappable object is created but it is never explicitly freed.

4 Results

The test programs shown in Figs 1 and 2 were run on a cluster of 10 Linux PCs each with 333MHz i686 Celeron processor and 32 MB physical memory. They were connected with Ethernet. Each PC was equipped with 2 GB local hard disk, where approximately 600 MB was available for our benchmarks. The LAM6.3b1 [12] has been used as a MPI implementation; the JVM was included in JDK1.1.7 for Linux. Each of the tests has been
run twice in C and in Java for two values of the parameter \( n \) (see Figs. 1 and 2). With \( n = 100 \) there were 100 internal iterations in each call of function \( f() \) and 1000 iterations for \( n = 1000 \).

Fig. 7 shows a small test case where both data and indices arrays are small enough to fit into the main memory (there are 3 arrays of 360360 double values each). Fig. 8 shows ooc case, where the indices array is large enough (3603600 double values) that it does not fit in main memory and must be stored on disk. Fig. 9 presents timings for a constant-time problem where both data and array sizes were proportional to the total number of computing nodes (sizes were 180180 \( \times N \) double values).

Java program execution times are about 2.5 - 3 times greater than the ones for C, nevertheless, we hope that the use of Java-to-native compiler would eliminate that difference. It is worth noticing that scalability of problem in C and Java is almost the same; and depends on the computations-to-communication ratio. For \( n = 100 \) there is noticeable saddle where communication and I/O costs override those of computations. It indicates that simple problems are often not worth parallelisation. It should be pointed out that our problem is a generic one, with random distribution of data so it requires a lot of communication each time data are localized.
5 Conclusions

In this paper, we have described some general issues of development of Java wrappers for native libraries using the Lip library as an example. Some new techniques of mapping Java features to native libraries has been proposed. We have demonstrated that Java may be used to compute irregular and out-of-core problems.

At present, we are developing a tool to support creation of Java wrappers to any native library. The main design goal is that the tool should generate the finest possible wrappers that could be done, as we would like to reduce “manual” part of the task as much as possible. We hope that the user should be able to give the wrapper specification, describing mapping native library objects (functions, structures, etc.) onto Java class hierarchy, together with the requirements on the function parameters, specification of error handling etc. Our tool will create the wrappers using that specification. The support for creation and modification with GUI will also be developed. The generator, which creates the starting-point specification automatically (in JCI style), is also under consideration.

Acknowledgments. We would like to thank Prof. Peter Brezany for valuable discussions. We are also grateful to Mr Tomasz Palac and Dr. Marek Valenta from the UCI AGH for their help. 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.

References


