Distributed High Performance Computing with OpusJava

Erwin Laure

Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
E-Mail: erwin@par.univie.ac.at

February 28, 1999

Extensive Paper for Section 3: System Software & Hardware Architectures

Extended Abstract

1 Introduction and Related Work

Advanced scientific applications are often built by composing existing software components that exploit specialized computing and algorithmic resources [6, 7]. One might e.g. extract data from a distributed data base, perform several numerical operations using a number of supercomputers, and visualize the results on specialized graphical workstations. We can identify a set of common properties of such applications:

1. different modules need to cooperate in solving a problem.
2. parallelism can be exploited within and across modules.
3. modules are written in different languages.
4. modules are dynamically instantiated on different, heterogeneous platforms.

Apart from these properties, it is often necessary to plug new modules into a running application on the fly or to change the hardware composition dynamically due to load balancing problems or hardware failures.

A number of systems have been developed which aim to support such applications. These systems range from low level services and middle-ware, like *Globus* [5], *CORBA* [10], and *PAWS* [1], to high level languages, like *Legion* [8] and to some extent *HPC++* [7]. However, these systems are either at a very low level and thus cumbersome to use, or lack efficient support for nested parallelism and high level coordination features.[1]

In this paper we present OpusJava, a Java interface to the coordination language Opus [3, 4]. OpusJava combines the strengths of Opus (especially the integration of task- and data parallelism) with the strengths of Java (especially the dynamic networking features). OpusJava provides a high level, uniform interface for distributed high performance computing. All low level details such as communication, data format conversion, or thread synchronization are completely transparent to the user. With help of OpusJava high performance modules written in HPF can be seamlessly integrated in larger distributed systems.

After a short review of the salient features of Opus we sketch the design of OpusJava and its components.

Opus was initially designed to support multi-disciplinary applications, providing an object-based extension of High Performance Fortran (HPF) that integrates coarse-grain task parallelism with HPF-style data parallelism. Its central concept is the *shared abstraction (SDA)*, which generalizes Fortran 90/HPF modules

[1] A profound discussion of related work is given in the full paper.

using an object-oriented approach and imposing monitor semantics. SDAs can be distributed, and thus internally data parallel, while task parallelism is exploited between different SDAs. SDAs communicate with one another via synchronous or asynchronous method invocation. A method's activation may be guarded by a logical condition clause. SDA objects are created dynamically on resources specified in an *on-clause* that may contain the name of a machine and the number of processors to be used. The execution of an Opus program can be thought of as a system of SDAs in which an SDA executes a method in response to a request from another SDA. A prototype of Opus is being implemented [9]. This prototype makes use of a multithreading runtime system for overlapping communication with computation and incorporates the *Vienna Fortran Compiler (VFC)* [2] for parallelizing the data parallel portions. Data exchange between SDAs is accomplished by exploiting highly optimized redistribution libraries which are invoked after a short hand-shake protocol between all the nodes of an SDA.

While Opus provides efficient support for coupling modules which exploit hybrid forms of parallelism on homogeneous systems (properties 1. and 2. from above), it lacks support for language interoperability and heterogeneous platforms (properties 3. and 4.). This is mainly due to the Fortran/HPF centric approach. OpusJava is designed to bridge the gap between efficient high performance computing and dynamic distributed computing.

2 OpusJava at a Glance

OpusJava is a Java package that provides Opus functionalities. Thus, it can be used to interface Opus (i.e. Fortran/HPF) applications with distributed Java (or C/C++ via JNI) applications or simply to program Java in a high level Opus style. The syntax of OpusJava is similar to Opus to provide a common look-and-feel.

Java was chosen as base language because of its excellent networking support. It provides platform independence, a common data representation, communication via RMI or sockets, multithreading, dynamic class loading, and reflection, each of which is indispensable for our applications in mind.

The main task of OpusJava is to provide both, access to Opus SDAs, and Java SDAs with their salient features. In doing so, OpusJava comprises three main components:

- **SDAServer**: The SDAServer is a daemon that is installed on every system which may participate in an OpusJava program execution. It exports RMI methods for both, creating and destroying SDAs.
- **SDAImpl**: The Java class SDAImpl is the base class for all SDA objects. While any user defined SDA needs to extend this class, the Opus compiler generates appropriate wrappers that extend SDAImpl for all SDAs in an Opus code automatically. SDAImpl implements all inherent SDA features like conditional method execution, storing of execution request, transmitting results, etc. Moreover, it exports RMI methods for synchronous and asynchronous method invocation.
- **SDA**: The final class SDA is in principle a pointer to SDAImpl objects. It contains a remote reference to its SDAImpl counterpart and provides methods for synchronous and asynchronous remote method invocations.

When using OpusJava all communication, RMI, JNI, and synchronization issues are completely transparent to the programmer; not even interface specifications via IDLs (which are required e.g. in CORBA) are necessary.

Let us illustrate the components and functionalities of OpusJava using an example (code fragments are given in Figure 1). Every module of an application needs to extend `SDAImpl`. In our case, it is a class `Test` which has one public method `foo`. If `Test` is an SDA from an Opus program, the corresponding class is generated by the Opus compiler. In such a case, `foo` would be a C wrapper for the original Fortran/HPF procedure which is accessible from Java via JNI (cf. Figure 2 which illustrates the invocation of an Opus method from Java).

In order to create/access an SDA object from a Java program, a new instance of class `SDA` needs to be created. The constructor has 4 arguments: the name of the SDA, an array of input arguments for the remote constructor, the machine-name on which it should be allocated, and the number of processors to be used. Internally, the constructor of `SDA` contacts the SDAServer on the given machine, passing along the request

```
class Test extends SDAImpl{

  /* constructor */
  public Test(){}

  /* method */
  public void foo(...) {...}
}
```

```
public class Main {
  public static void main(...) {

    /* create new instance of Test */
    SDA bar = new SDA(``Test'',null,''machine'',1);

    /* invoke a method of Test synchronously */
    Object[] args = ...
    bar.call(``foo'',args);

    /* invoke a method of Test asynchronously */
    Event ev = bar.spawn(``foo'',args);

    /* synchronize */
    ev.wait();
} }
```

Figure 1: OpusJava Example

for creating the SDA on the given number of processors with the given input arguments. The SDAServer exploits Java's reflection mechanism to check, whether appropriate classes are available. If not, dynamic class loading can be used to load the classes from user specified resources on demand. After the SDA object has been created, the SDAServer registers the new object in the RMIRegistry and sends back the registered name. The `SDA` object creates a remote reference to the registered object via which all subsequent method calls occur.

Class `SDA` provides methods for calling remote methods in a synchronous (`call`) or asynchronous (`spawn`) way. In the first case, the call is blocking until the results are available, while in the latter case an `Event` object is returned immediately, which can be used for synchronization later on (e.g. with the method `wait`). When invoking a method, the method name and an array of input arguments needs to be passed. On the recipient side, the base class SDAImpl provides mechanisms for receiving and evaluating the validity of requests utilizing Java's reflection mechanisms.

3 Conclusions

Let us summarize the salient features of OpusJava: OpusJava provides a high level interface for coupling and coordinating modules written in different languages and exploiting multiple levels of parallelism. High performance components can be seamlessly and efficiently integrated in a bigger system by calling Opus from OpusJava, which is enabled by appropriate wrappers generated by the Opus compiler. These high performance components need not to be changed and can still exploit highly optimized communication mechanisms available on their target platform. The user of OpusJava can focus on the algorithmic problem:

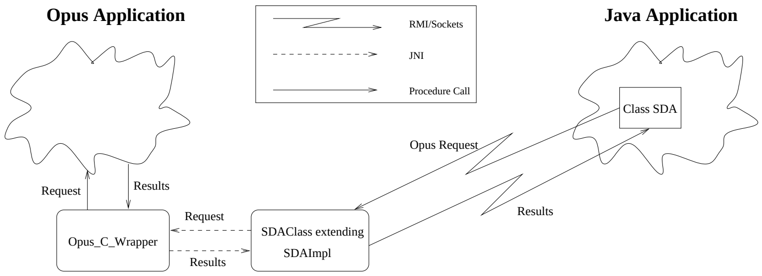

Figure 2: Opus/Java Interaction with OpusJava

all low level details like communication, RMI mechanisms, data format conversions, or JNI are transparent. Moreover, OpusJava employs Java's dynamic class loading features, which allow new modules to be plugged into a running application dynamically.

In the full version of the paper, the OpusJava architecture and its interface to Opus are elaborated in more detail. Moreover, runtime results will be presented which compare the method invocation overhead of OpusJava with that of pure Opus.

References

[1] P.H. Beckman, P.K. Fasel, W.F. Humphrey, and S.M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proc. HPDC*, Chicago, IL, July 1998.

[2] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[3] B. Chapman, M. Haines, E. Laure, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus 1.0 Reference Manual. Technical Report TR 97-13, Institute for Software Technology and Parallel Systems, University of Vienna, October 1997.

[4] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6/9:345–362, Winter 1997.

[5] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[6] I. Foster and C. Kesselman, editors. *The Grid.* Morgan Kaufmann, 1999.

[7] D. Gannon et al. Developing Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science & Engineering*, April-June 1998.

[8] A.S. Grimshaw, W.A. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[9] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna, to appear.

[10] OMG. *CORBA/IIOP 2.2 Specification*, June 1998.