6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Compiling P–GRADE programs for the JGrid architecture[*]

László Lövei

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest
e-mail: lovei@elte.hu

Abstract

Grid systems are large, geographically distributed computational environments. JGrid is a Grid infrastructure, that provides a software environment for such systems. It is built on top of the Java-based Jini technology, which provides a common platform and a service-based lookup system.

P–GRADE is an interactive, graphical software development tool, designed to build distributed parallel programs using message passing for communication. It is based on the GRAPNEL hybrid language, which uses graphics to describe communication topology, main control structures and communication actions, while sequential computations can be written in an arbitrary programming language.

P–GRADE originally supported the C language using PVM or MPI message passing libraries. This paper gives a description of a compiler package, which implements the GRAPNEL language using the JGrid system as a communication infrastructure, and provides a compiler tool that translates P–GRADE programs to the Java language using this implementation.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors - *code generation, compilers*; D.1.3 [Programming Techniques]: Concurrent Programming - *distributed programming*

Key Words and Phrases: GRID, Java, Jini, P-GRADE, compiler

1. Introduction

The JGrid project[1] aims to develop a Grid middleware using the Java-based Jini[2] architecture, and to provide a graphical integrated development environment to support writing distributed applications using this middleware. The P–GRADE

[*]Supported by the Hungarian Infocommunications Technologies and Applications Programme (IKTA), Grant Nr. 089/2002

environment[3] has been selected as an IDE, which supports not only application development, but debugging, monitoring and analyzing as well. To adapt P–GRADE to JGrid, the main task is to enable the translation of P–GRADE programs to the Java language using the services of JGrid.

1.1. P–GRADE

P–GRADE stands for Parallel Grid Runtime and Application Development Environment. Its main goal is to provide an integrated set of programming tools for development of distributed programs using the message passing paradigm. P–GRADE uses the Graphical Process Net Language (GRAPNEL). In fact, it is a hybrid language, as it defines textual program parts as well as graphical structures.

Distributed programs written in GRAPNEL consist of three design levels, these are called application, process and text level.

Application level. In this level the communication structure is given by a graphical notation. *Processes* and *process groups* can be defined, every one of them can be assigned *communication ports*, and these ports can be connected by *channels*. Every channel has set of *protocols* assigned to it, which describes what kind of data can be sent through the channel. An example of this level is shown in figure 1.

Process groups contain processes and other groups. These are given by the same application level notation. There are special groups with predefined communication structures, these are called *communication templates*. Predefined templates are the Farm, Pipe and Mesh templates.

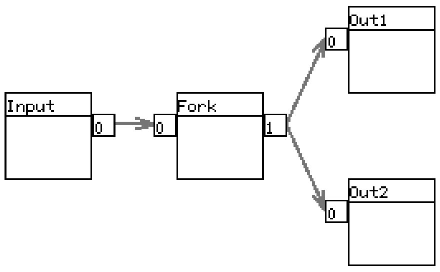

Figure 1: Application level of a P–GRADE program

Process level. Every process contains communication operations, which send or receive messages through the ports connected to the process. This level specifies an algorithm skeleton for the process, which contains every communication operation. It is described graphically, as a program graph, which has loop and conditional constructs, and contains sending, receiving and sequential steps.

Text level. Sequential step mentioned at the process level are arbitrary computations, which do not contain any communication. These computations are described in a traditional programming language, in our case, Java. These program parts are given in plain text, and should be copied verbatim into the resulting program.

Sometimes these parts need information about the state of the P–GRADE program. This is supported by functions defined in GRAPNEL. These functions return various information about the communication structure and state of the calling process.

1.1.1. Types of communication

The main role of P–GRADE from the view of application developers is that P–GRADE connects the parts of the programs by graphically described communication operations. As these operations are the "heart" of P–GRADE, there are many possibilities in them.

Point to point communication. Point to point message sending and receiving is the most simple communication action. Receive operations always block the process, and wait for an incoming message. Sending operations can be synchronous or asynchronous.

Group communication. When more than one partner or a group is connected to a communication port, every partner can be addressed in one step. This is called *group communication*. Messages can be sent to a group using *multicast* semantics, which means that the same data is sent to every partner, or *scatter* semantics, when every partner is sent different data. Receive semantics is called *gather*, which means that one message is received and stored from every partner.

Alternative receive. There is one special kind of receive operation called *alternative receive*. This operation can use more than one ports and groups as message sources, and receives exactly one message from them, the first one that is available at the time of the operation.

1.2. The JGrid architecture

JGrid is a Grid middleware that provides an infrastructure to connect a large number of computers and use their resources as if they were a single virtual computer. It supports different architectures by the use of the Java platform, which hides the architectural differences by defining a common Java Virtual Machine. Resources in JGrid are represented by *services*, which can appear or disappear at any time. This approach provides a dynamic and scalable architecture. which is essential in Grid systems.

The Jini technology. Services are supported by the *Jini* technology in JGrid. It is an open network architecture designed to help client programs to find and use services in a dynamic environment, and makes a good base for a Grid system[4]. Jini automates the service finding process by providing a *Lookup Service*, which keeps track of available services. Every client and service can *discover* a Lookup Service using a special discovery protocol. This protocol uses IP multicast, so it is possible to automatically discover distant parts of a Jini network.

A client can look for a service that implements an arbitrary interface. It can select one or more from the search results, and use it (or them). A client that uses a service gets a *lease*, which expires after a given time. A lease can be renewed, so it does not limit the usage of a service, but when one of the parties fails, the lease expires, and a notification is sent, so the failure can be detected.

Services of JGrid. The JGrid system is a set of high level Jini services. The most important service is the *Compute Service* that provides a computational resource. It runs Java programs using the resource, this way exporting a Java Virtual Machine into the Grid. Every program to be run must implement a Java interface called Task. This interface is used to provide a context for the program, and to execute the main method of the program in a thread of the Virtual Machine. A set of programs can communicate with each other using the context, message sending operations are available for this purpose. The JGrid system supports a Java binding of the standard Message Passing Interface (MPI, see [5]).

Other services include a *Storage Service*, which provides disk space to store data in a virtual filesystem; a *Broker Service*, which manages load balancing and selecting the most appropriate resources for a task; and a *Grid Access Point*, which handles authentication and authorization. These are important services, too, just they are less significant in the P–GRADE compilation process.

2. Code generation

Our task is to generate Java code from a P–GRADE program, using the services of JGrid. The most elegant approach to this is creating a *library* that supports every language element of GRAPNEL. This way, generated code will be quite simple and less error-prone. The other advantage is that the interface of the library can be independent of the underlying communication architecture, so porting to another communication system requires modifications in the library only.

As we have seen, in JGrid, every process is a Java object that can be run using the Task interface. The library can support this by providing its interface via *inheritance*. Every P–GRADE process class must be derived from a generic process class, which implements the Task interface, and hides the details of the JGrid system. In Java, requiring the use of inheritance is considered obtrusive, because every class can have only one superclass, but as we want to provide an interface for a generated code, which has no other purposes than implementing a P–GRADE program, this limitation has no drawbacks here.

Using inheritance has one more advantage. The sequential code that is given textually in GRAPNEL, uses function calls to gather information about the environment. These functions can be implemented as methods in the generic process class, and this way they can be used directly by the sequential code, which will be copied into the code of the derived class.

Every design level of P–GRADE needs its own library support. The application level consists of process groups, which manage the communication topology of the program. There is a special group, that contains the top level processes and groups of the application, and contains the code that starts the whole program. The process and the text levels has their support in the generic process class, which supports the communication operations directly, and provides methods for the textual code.

2.1. Application level code

Application level is mainly responsible for building the communication topology of the processes. The support class has methods for defining process and group instances, and for connecting their ports. An example of an application level generated code is shown in figure 2.

```
public ForkApp() {
    super(4);
    createProcess(PROC_Input, new Input_code());
    createProcess(PROC_Fork, new Fork_code());
    createProcess(PROC_Out1, new Out1_code());
    createProcess(PROC_Out2, new Out2_code());
}

protected void init() {
    connectPorts(PROC_Input, Input_code.PORT_0,
        PROC_Fork, Fork_code.PORT_0);
    connectPorts(PROC_Fork, Fork_code.PORT_1,
        PROC_Out1, Out1_code.PORT_0);
    connectPorts(PROC_Fork, Fork_code.PORT_1,
        PROC_Out2, Out2_code.PORT_0);
}
```

Figure 2: Generated code of the application in figure 1

In GRAPNEL, processes and groups are referred to by their names. In the other hand, the implementation is better with using sequential integer identifiers. Even port numbers are bad for this purpose, because they might not be assigned strictly sequentially. So, processes and groups define constant names for processes, groups and ports contained in them. The identifiers in the example, like PROC_Input or PORT_0 are such ones. The classes, like Input_code, has their names derived from the real process names, and contain the code of the process. The library calls in this level are quite straightforward, every call matches an element in the P–GRADE program (see also figure 1).

2.2. Process and text level code

The generated Java code of the processes contains the implementation of the communication operations of the process design level and the textual code from the text level. The latter is the simpler, its "generation" involves copying only. The former is more complicated, it requires the implementation of the program graph and the communication operations.

A program graph contains conditional and loop nodes, these can be implemented by simple `if` and `while` statements. Sequential nodes are textual program blocks, which are copied verbatim. Communication operations are more complex, not only because there are many types of them, but there is the problem of *protocols:* the protocol interpreter code cannot be implemented in the support library, it must be generated individually for every protocol.

```
initSend(PORT_0, 0, true);
pkint(count);
for (int __pr=0; __pr<3; ++__pr)
    pkdouble(data[__pr]);
send(0);
initRecv(PORT_0, 0);
recv(0);
count = upkint();
data = new double[3];
for (int __pr=0; __pr<3; ++__pr)
    data[__pr] = upkdouble();
```

Figure 3: Generated code for message sending and receiving

An example of a simple protocol is shown in figure 3, as part of a point to point sending and receiving operation. The protocol consists of an integer and a triplet of floating point data. The communication action in the example uses the variables `count` and `data` to store data. Interpreting the protocol means storing or retrieving data to or from the message buffer. Communication of type *scatter* or *gather* involves using of array element access at each variable, as using them requires storing different data for every partner.

Group communication is implemented by point to point operations, as shown in figure 4. When different members of a group should receive different data, there is no other way to perform the operation.

Alternative receive is a special case, because different partners may use different protocols, and data may have to be stored in different variables. Protocol interpreter code for alternative receive operations therefore consists of conditional blocks. Each protocol that appears in the operation has a conditional branch that interprets that protocol, this way the correct interpreter code is selected during run time.

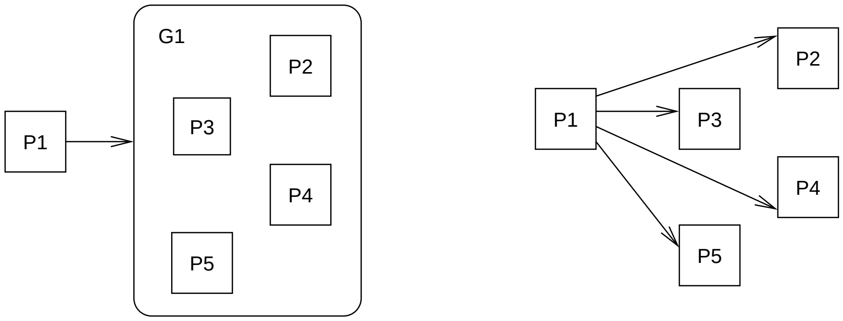

Figure 4: Implementation of group communication by point to point operations

3. Library implementation

As we said before, the task of the support library is to provide an easy-to-use interface for the generated code by implementing GRAPNEL language elements. The interface should hide every JGrid-specific aspect of the implementation, so that generic Java code can be generated using the library.

There are two main areas where a real JGrid implementation is needed for the abstract interface: the starting of a program and the communication. Every other element of GRAPNEL can be implemented by this core (like communication templates and protocol interpretation) or does not need any library support (like loops or conditionals in processes).

3.1. Program startup

The startup phase of a program begins when the application is started on a JGrid client machine, and ends when the processes are ready to be run at the Compute Service nodes. Process objects are created by the generated application code, so library support is only needed for communication channel definition and process distribution among the nodes.

As the process objects will be sent through the network, they cannot have any Java references to each other. This means that channel definition must be done using primitive values. Every process object gets a communication ID; JGrid supports the MPI standard, in which this ID is an integer sequence number. A channel endpoint is defined by this ID and a port number. During the channel definition phase of the startup, these endpoint identifiers must be distributed among the processes.

When every process knows its partners' identifiers, they can be sent to the nodes where they will be run. This is done by the JGrid system, the only task of the library here is to provide an abstract interface for this operation, that is, the application program just have to call a method to distribute and start the processes.

3.2. Communication

The communication interface of the support library provides a message buffer, and methods to fill it with data, read data from it, send the buffer to other processes, and to receive data from other processes into the buffer.

The methods that handle the contents of the buffer are implemented by the standard Java serialization routines. P–GRADE requires only primitive data to be sent, but this approach gives the possibility to transfer any serializable object between the processes.

In P–GRADE, every channel can be assigned more than one protocols. These protocols have an integer identifier in P–GRADE, which can be used by the implementation to make a distinction between messages of different protocols. The MPI standard defines a so-called *message tag*, which can be used to filter messages coming to a process. This is needed to implement a receiving operation which accepts data of only one protocol.

Data transfer. As we have seen, every communication operation can be implemented by point to point sending and receiving. So it seems that simple point to point MPI operations are enough to implement the support library. Every send and receive operation can be done by an asynchronous, blocking MPI send or receive call – except alternative receive.

Alternative receive can be implemented in MPI, but it requires non-blocking receive operations. A non-blocking MPI call returns a request identifier, which can be checked later to see whether the operation has been finished or not. It is possible to have a set of requests, and wait for any of them to complete. This matches the alternative receive exactly. It has only one problem: when more than one source sends data, still only one of them is processed. Requests which are not processed, must be saved and used in the next receive operation, even if it is a point to point operation.

The solution is to use non-blocking MPI operations everywhere. Every communication partner has a pending request associated with it, and every receive call uses these requests. A point to point operation uses only one, an alternative receive uses more; after the operation, the completed request is restarted to make sure that the next receive operation can use a valid request.

4. Summary

The JGrid project aims to build a service-based Grid infrastructure. Besides creating the services, it is important to provide tools for the users of the infrastructure, including support for application development. P–GRADE is well suited for this task, as it covers one of the main areas of Grid application development, distributed programming.

P–GRADE must have been adapted to JGrid. The main task in the adaptation was the creation of the compiler that translates GRAPNEL programs to Java. Code generation was designed to be as easily maintainable and extendible as possible. It was achieved by creating two layers: a support library and a code generator. The latter outputs generic Java code, which is independent of the infrastructure it runs on. The library contains the implementation details, it can be much easily modified for other platforms than the code generator.

In the JGrid project both the library and the code generator has been implemented and tested. Experiences of the development and testing have shown that we have succeeded in creating a working and easily extendible compiler package.

References

[1] Z. Juhász, R. Lovas, P. Kacsuk: *JGrid: A Jini-based Service Grid,* IEEE International Conference on Cluster Computing (Cluster 2003), pp. 28-30.

[2] Jini Network Technology
http://www.jini.org

[3] P. Kacsuk, G. Dózsa, R. Lovas: *The GRADE Graphical Parallel Programming Environment,* Parallel Program Development for Cluster Computing, Methodology, Tools and Integrated Environments, Nova Science Publishers, Inc. pp. 231-247, 2001.

[4] Sz. Póta, K. Kuntner, Z. Juhász: *Jini network technology and Grid systems,* Proc. MIPRO 2003, Hypermedia and Grid Systems, Opatija, Croatia, 19-23. May 2003., pp. 144-147.

[5] Message Passing Interface Standard 1.1
http://www.mpi-forum.org/docs/mpi-11.ps

[6] Zoltán Csörnyei: *Compiler algorithms,* Erdélyi Tankönyvtanács, Kolozsvár, 2000 (in Hungarian).