COARSE-GRAIN PARALLELISM USING REMOTE METHOD INVOCATION

Alain Hébert
École Polytechnique de Montréal
C.P. 6079 succ. “Centre-Ville”
Montréal Qc. CANADA H3C 3A7
alain.hebert@polymtl.ca

Abstract

The paper describes a user–oriented framework specifically designed to facilitate the development and supervision of coarse-grain parallel applications in reactor physics. The proposed user–oriented framework was designed and implemented in Java, in such a way to obtain a simple and robust application. The proposed approach is based on *Java Native Interface* (JNI) for integrating the Fortran legacy code and on *Remote Method Invocation* (RMI) for distributing the calculation load over the farm of processors. Dynamic code downloading over the network is possible.

We are presenting the application of this approach to supervision of lattice calculations using the open source Dragon code. The Java layer surrounding Dragon can also be used to construct execution procedures, computational schemes and graphical user interfaces. This approach can be used with any existing legacy Fortran code, as soon as its input/output data structures are Dragon-compatible.

Introduction

Many fields of reactor physics are suitable for a coarse-grain parallel architecture involving asynchroneous execution of code components. In this paper, we have investigated the field of lattice calculations, where the neutron transport equation must be solved hundred of times, for increasing values of the fuel burnup and for varying global parameters such as fuel temperature and coolant density.

Without a coarse-grain parallel execution, all these calculations must be executed sequentially, leading to very long job restitution time. Significant improvement of the restitution time can be obtained by executing asychroneously the calculation work over a farm of workstations, leading to high performance computing. However, this operation is difficult if the legacy code is written in Fortran. Moreover, coarse-grain parallelism can lead to synchronization problems when the same computer ressource is accessed by parallel threads of execution.

The design of a nuclear reactor involves the combined use of computer codes belonging to many disciplines: reactor physics, thermalhydraulics, mechanics, fuel thermo-mechanics, etc. Most of these codes have been programmed in Fortran in the seventies, are still in production and are likely to be used for the design of next generation nuclear reactors. Reprogramming them in a modern object-oriented language such as $C++$ would be a very risky operation: programming many million lines of code cannot be done over a short period of time, the new codes would be more buggy than the original ones and the execution speed would be reduced. However, modernization is required, as

- computers have been improved over the recent years, namely with the availability of multi-processing, coarse-grain parallelism and remote supervision techniques. These improvements make possible the resolution of problems which were considered unsolvable by the previous generation of computers.
- the design work is an iterative process which can be speeded by an efficient user interface.
- the requirement for quality assurance (QA) procedures has increased. We need to apply revision control not only on the code source, but also on the user procedures and computational schemes.
- graphical user interfaces are required by end users.

This paper is related to the conception of a coarse-grain supervisor named Jargon (standing for *Java Remote Dragon*) .[1] The design of Jargon was mainly directed to provide an efficient solution to the coarse-grain parallelism and remote supervision problems. We have chosen to design Jargon in Java, mainly in a search for the simplest approach which can fulfill each of the above-mentionned needs.

Description of the Coarse-Grain Supervision Approach

The Jargon coarse-grain supervisor is based on a three-level architecture. The first level include the Fortran modules (or operators), used without modification from the official version of Dragon[2]. The second level include a hierarchy of Java classes implementing user procedures, computational schemes and graphical user interfaces (GUI). The third level is the end-user interface, based on BeanShell scripts[3] and GUI instances.

Integration of the Fortran Code into Java classes

In the proposed approach, the legacy Fortran code (namely Dragon) is embedded into a Java class and each module of Dragon (now called *operator*) is a *native* method call of this class. The CLE-2000 supervisor[4] is therefore removed and replaced by Java. Java is intended to perform the following tasks:

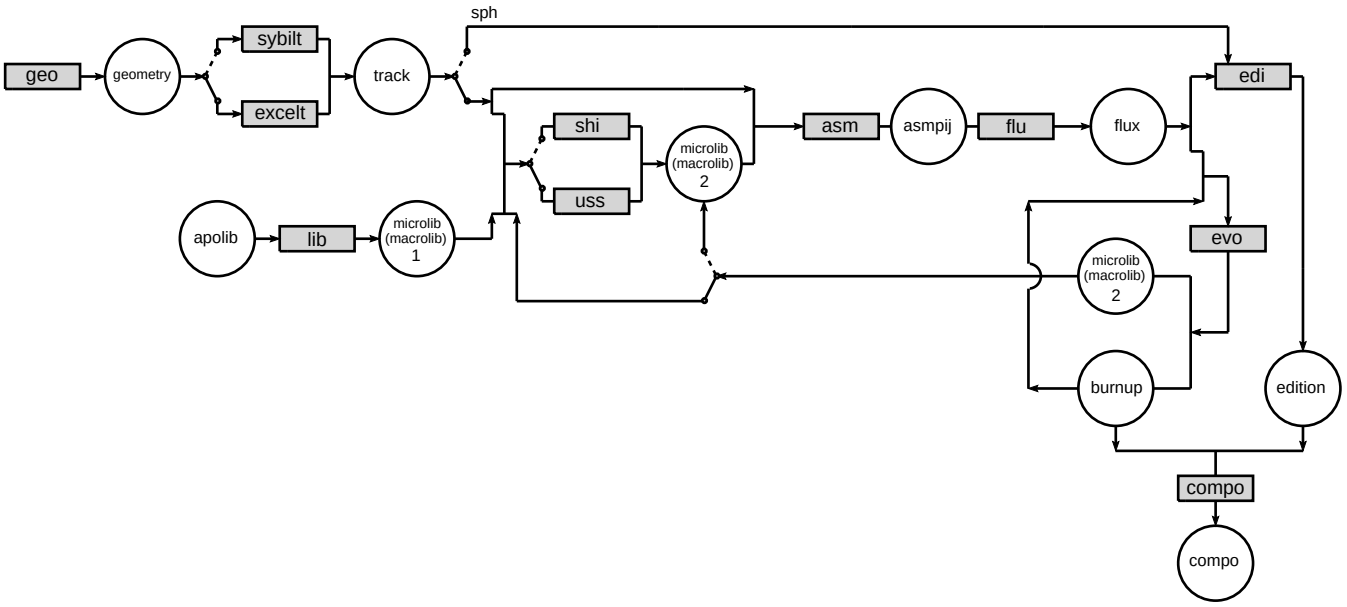

Figure 1: Flow diagram of Dragon.

- to set the data flow of the Dragon data structures[5] (geometry, macrolib, flux, compo, etc) between the operators (library access, self-shielding, flux and leakage calculation, homogenization/-condensation/equivalence, etc.). The Dragon data flow is represented in Fig. 1 where the data structures and operators are represented by circles and rectangles, respectively. Here, we use Java to implement the data flow corresponding to a particular *computational scheme*.
- to write the calculation procedures (burnup, reactor database construction, etc.)
- to write the computational schemes, adapted to a particular reactor design
- to build a graphical user interface (GUI).

This approach is highly-compatible as the Dragon data structures are associative tables and heterogeneous lists, implemented in the LCM application programming interface (API). These data structures are similar to the container legacy classes available in Java (`HashTable` and `Vector`). We have therefore reprogrammed the LCM API[6] in *C* in such a way to keep the same behaviour as the Ganlib Fortran version. Next, we have implemented the LCM bindings with Fortran and Java. The LCM Java binding are available as a `Jlcm` class with access methods implemented using *Java Native Interface* (JNI). The resulting architecture is represented in Fig. 2. This work was performed following the terms of the GNU Lesser General Public License (LGPL).

Next, we have developed a hierarchy of Java classes to facilitate the writing of procedures, computational schemes and data files. An UML representation of this object model is shown in Fig. 3. Its main characteristics are the following:

- each Dragon operator (aka module) is mapped onto a specific class that inherit from a generic class named `Operator`. This generic class is responsible for the transformation of any operator class attributes into a valid Dragon input data file.

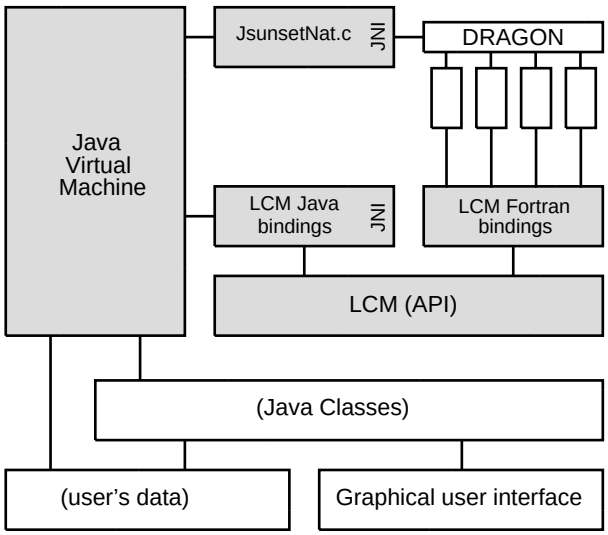

Figure 2: Java–Based Scripted Dragon architecture.

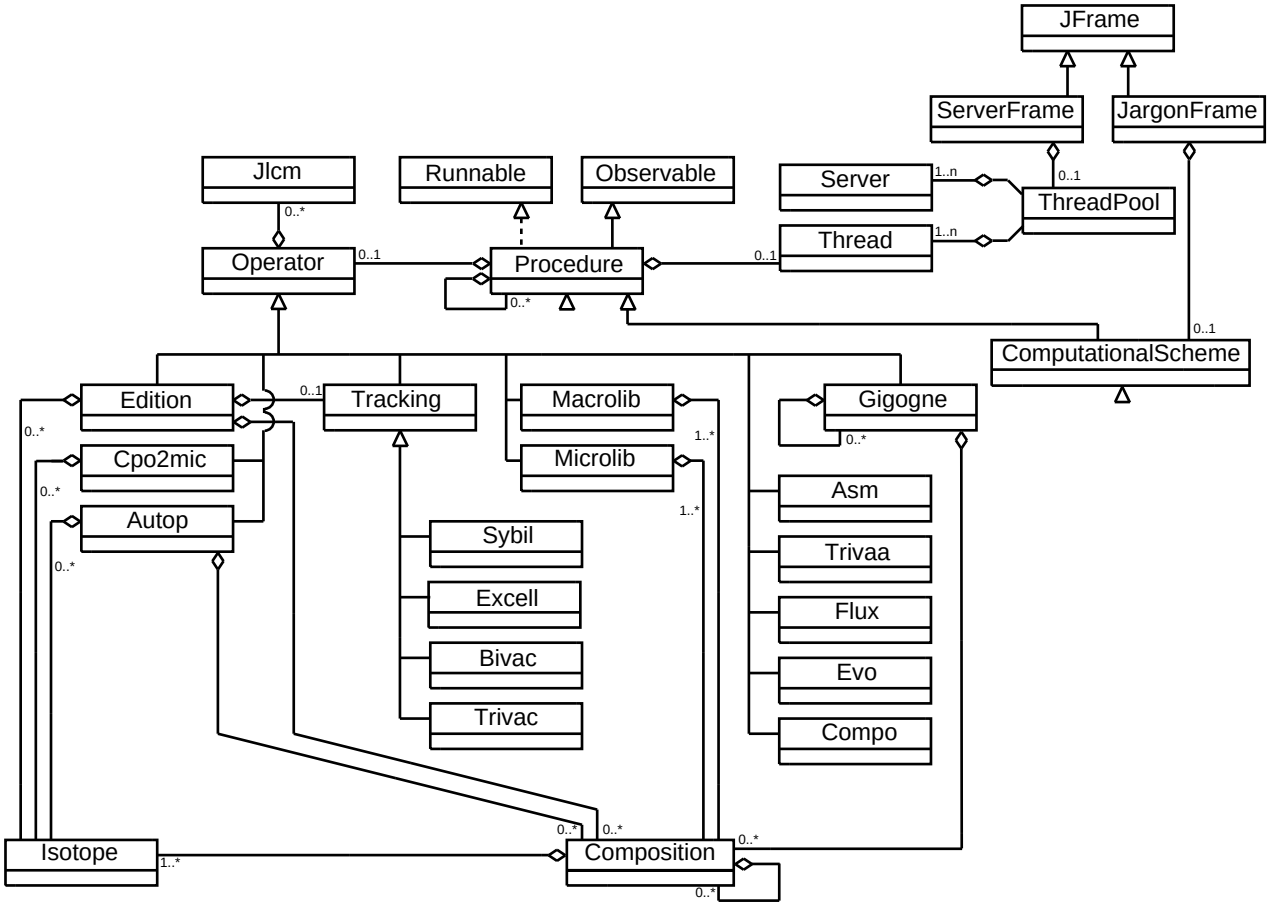

Figure 3: UML diagram of the user-oriented object model.

- the Dragon data belonging to a specific operator is given as instance variables of the corresponding `Operator` object.

- the LCM data structures produced by this operator are embedded into `Jlcm` objects. The references to these `Jlcm` objects are stored as instances variables of the `Operator` object.
- a generic `Procedure` class is available as a template to construct procedures. The `Procedure` class has the capability to execute asynchroneously on a remote computer.
- a generic `ComputationalScheme` class is available as a template to construct computational schemes. The methods of a `ComputationalScheme` class are automatically detected by the Jargon GUI using the reflection mechanism of Java.

Design of the Remote Execution Capabilities

The remote execution capabilities has been implemented as a *distributed object* model and implemented using the *Remote Method Invocation* (RMI) mechanism of Java. RMI applications are comprised of two separate programs: a server and a client. The server application is simply the following Java class, capable of executing any procedure `t` implementing the interface `ProcedureIntf`:

```
public class ProcServerImpl extends UnicastRemoteObject implements
  ProcServerIntf {
  public ProcServerImpl() throws RemoteException {
  }
  public Operator executeProc(ProcedureIntf t) {
    return t.exec();
  }
}
```

where the procedure `t` belongs to the client's *codebase* and where the `Operator` object returned by `executeProc` and by the `exec()` method of `t` is containing the result of the remote calculation. Output `Jlcm` objects may be embedded in this `Operator` object. Note that the server has no initial knowledge of the procedure `t` it is going to execute. The *bytecode* of `t` is downloaded dynamically from the client's *codebase* to the server using a mechanism called *Dynamic Code Downloading* (DCD) which is one of the most significant capability of the Java platform. DCD has the ability to dynamically download Java code from any *Uniform Resource Locator* (URL) to a Java virtual machine (JVM) running on the server.

On the client's side, the procedure `t` must extends the `Procedure` abstract class, which impose two implementation conditions:

- The `exec()` method must be implemented by the procedure. The `exec()` method is intended to be executed remotely.
- The `exec()` method must use serializable objects as input and return one serializable `Operator` object as output. The input and output objects are instances variables of the `Procedure` object.

It should be noted that `Jlcm` objects are serializables in spite of the fact that they contains native fields and methods. However, the native implementation of these fields and methods must be available in the server. The environment variable `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH` must therefore be set on the server to include the native code required by `Jlcm`. These environment variables must also include the native code of the legacy Fortran application (Dragon in our case). The serialization of the native fields present in a `Jlcm` object is based on specific implementations of methods `readExternal` and

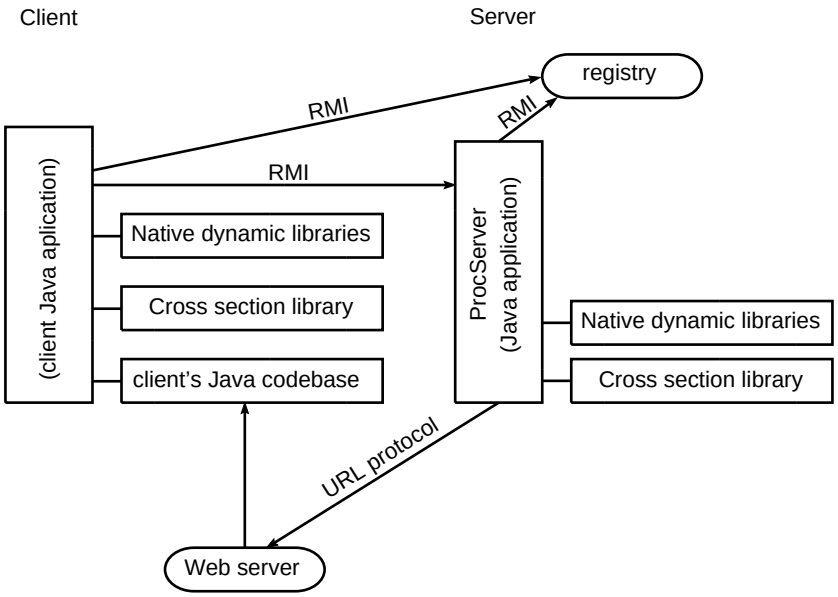

Figure 4: Dynamic code downloading and remote execution.

`writeExternal` where the data stream is compressed during its transition over the network. Jargon would be useless without compression due to the communication overhead associated with ASCII streams.

The overall communication process of a remote calculation is represented in Fig. 4 and proceed as follows:

- The RMI client request a reference to the `ProcServer` remote class from the server *registry*.
- All the input objects required by the remote calculation are automatically serialized by the RMI mechanism and transferred to the remote application.
- The server application detect a call to a `Procedure` method belonging to the client's Java *codebase*. The bytecode of this method is downloaded from the URL location of the *codebase* to the remote application.
- The server execute the `exec()` method of the `Procedure` and create an output `Operator` object containing the result of the remote calculation.
- The output `Operator` object is serialized and sent back to the client using the RMI mechanism.

Design of the Asynchroneous Execution Capabilities

Asynchroneous execution is the ability for a client to send request to many servers, to execute many remote calculations in parallel and to synchronize their results on the client. This capability is design around the *multithreaded* programming support of the Java platform. The main difficulty here is the synchronization of the resources access when two threads are executing simultaneously on the same server. A mechanism must be implemented to avoid the simultaneous access by parallel threads to a single LCM object or to the cross section library.

The initial design is represented in Fig. 5. In this approach, a new `Thread` object is created for each remote calculation without taking into account the number of servers. As a result of this design, many remote calculations are executing in parallel on each server, leading to synchronization difficulties.

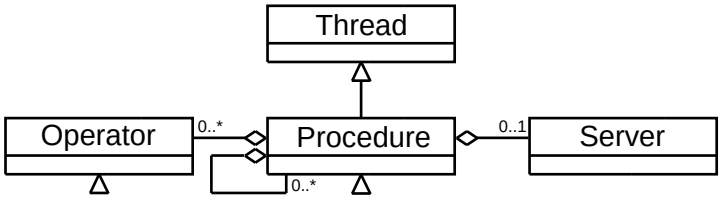

Figure 5: The initial multithreading design.

An improved design is represented in Fig. 6. In this case, we create a `ThreadPool` object as described in Ref. 7. This object is containing n deamon `Thread` objects respectively connected to n servers. The `ThreadPool` object is also containing a `LinkedList` object containing a queue of the remote calculations to be executed. Using this design, a server is always executing a single remote calculation, avoiding synchronization problems. Morever, the calculation load is better dispatched on the servers in cases where some remote calculations are longer than others. The synchronization of the remote calculations on the client are possible through methods `join()` and `isAlive()` belonging to all `Procedure` object. The recovery of a remote calculation is performed by implementing the `update()` method of the `Observer` interface.

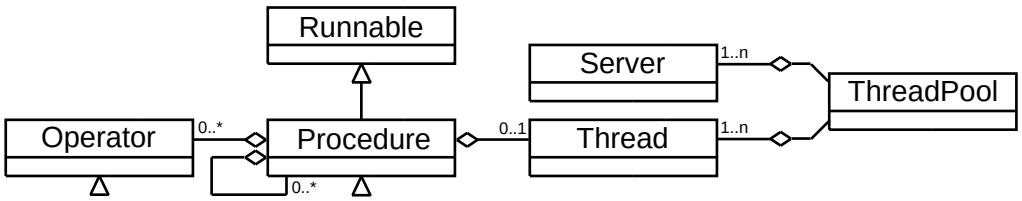

Figure 6: The actual multithreading design.

Results

The execution of any Dragon operator proceed in two steps:

- construction of an operator instance of the required type and definition of its instance variables
- execution of this operator instance using the `exec()` method belonging to any subclass that inherit from the `Operator` class. Note that the execution of the `Operator` instances, given as input to a `Procedure` instance, can be performed by the `Procedure` class itself. This help to simplify the user scripts, as presented in the following example.

As an example, consider a simple Dragon input data file corresponding to the solution of a non-depleting Wigner-Seitz unit cell. The corresponding BeanShell script is written

```
import jargon.*;
myGeom = new Gigogne("myGeom", "CARCEL", new int[]{2});
myGeom.bc = new String[][] {{"X-","REFL"},{"X+","REFL"},{"Y-","REFL"},
                            {"Y+","REFL"}};
myGeom.meshx = new float[] {0.0f, 3.6f};
```

```
myGeom.meshy = myGeom.meshx;
myGeom.radius = new float[] {0.0f, 0.829f, 1.029f};
myGeom.mix = new int[] {1, 2, 3};

myMacro = new Macrolib("myMacro", 1, 5, 1, 1);
// mixture 1:
myMacro.xs[0].total = new float[] {0.3652f};
myMacro.xs[0].nusigf = new float[] {0.05564f};
myMacro.xs[0].chi = new float[] {1.0f};
myMacro.xs[0].scat[0].xscat = new float[] {0.3234f};
myMacro.xs[0].scat[0].njj = 1;
myMacro.xs[0].scat[0].ijj = 1;
// mixture 2:
myMacro.xs[1].total = new float[] {0.4029f} ;
myMacro.xs[1].scat[0].xscat = new float[] {0.4000f} ;
myMacro.xs[1].scat[0].njj = 1;
myMacro.xs[1].scat[0].ijj = 1;
// mixture 3:
myMacro.xs[2].total = new float[] {0.3683f} ;
myMacro.xs[2].scat[0].xscat = new float[] {0.3661f} ;
myMacro.xs[2].scat[0].njj = 1;
myMacro.xs[2].scat[0].ijj = 1;

myScheme = new SimpleCell("myScheme");
myScheme.setGigogne(myGeom);
myScheme.setMacrolib(myMacro);
myScheme.setEdit(1);
myScheme.setOption("K");
myScheme.run();
```

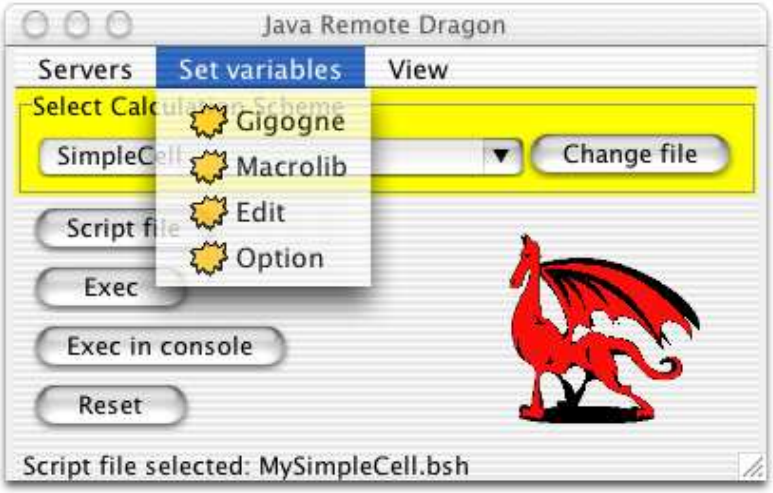

Figure 7: Graphical User Interface for Jargon execution.

In the previous example, the `ComputationalScheme` class `SimpleCell` was instanciated and run using a BeanShell script. It is also possible to execute the same problem using the generic Jargon

GUI, as shown in Fig. 7.

We will now present a test case involving remote execution and coarse-grain parallelism. We consider a lattice calculation of the 17×17 pressurized water reactor (PWR) assembly shown in Fig. 8. The fuel burnup is increased from 0 to 500 MW-Day/Tonne in 5 steps. At each burnup step, branch calculations are performed for 3 values of the water temperature and 2 values of the fuel temperature. A total of 30 branch calculations are sent to remote servers using the RMI technique presented above. Each elementary calculation involve a complete resonance self-shielding calculation followed by a multigroup flux calculation. Exact 2D collision probabilities are used for self-shielding and flux calculations. The overall procedure is set in the `SimpleCompo` computational scheme.

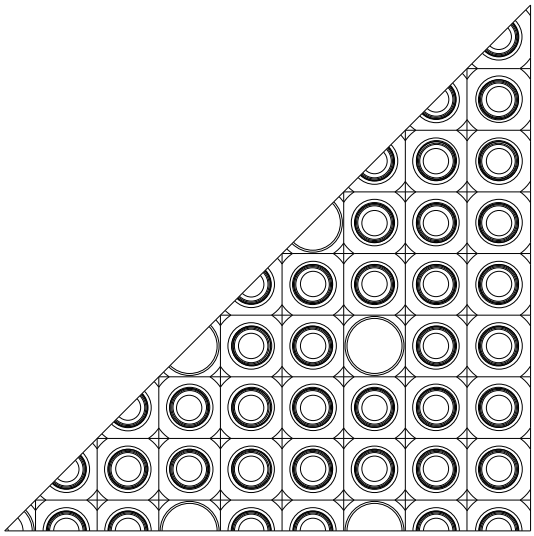

Figure 8: Description of the PWR assembly.

The thread pool is set from a BeanShell script using the following input:

```
boolean remote = true; // set true for remote execution.
pool = null;
for(int i=0; i<bsh.args.length; i++) print("server " + i + ": " + bsh.args[i]);
if (remote) pool = new ThreadPool(bsh.args);
```

where `bsh.args` is a BeanShell system variable containing an array of Strings passed as command line arguments and containing the names of the servers. The thread pool can also be set using the *Servers* menu item of the Jargon GUI, as shown in Fig. 9.

The `SimpleCompo` computational scheme is executed using a client and 1 to 4 servers. All these CPU are IBM pSeries 630 (1Ghz) processors. The calculation time required to produce the 10095 kilobyte reactor database (the Compo object) is plotted against the number of servers in Fig. 10. The maximum theoretical value shown on this curve is the calculation time requires with vanishing communication overhead. We observe CPU savings consistent with the maximum theoretical value.

Conclusions

We have successfully developed a Java-embedded version of Dragon featuring modern programming characteristics not available in Fortran. This paper is related to the remote execution capabilities brought by various mechanisms available in the Java platform. A three-level architecture was set to permit the

Figure 9: GUI selection of the thread pool.

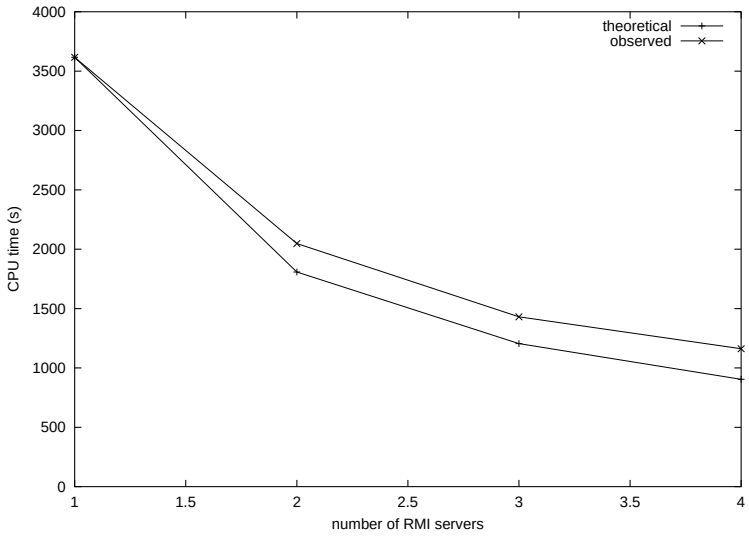

Figure 10: CPU saving using the thread pool..

asynchroneous execution of procedures on many servers and coarse-grain supervision. This capability is based on the *Remote Method Invocation* and *Dynamic Code Downloading* mechanisms of the Java platform. Synchronization and load balancing on the servers was accomplished using a thread pool design. The RMI mechanism was adapted to compress and to carry the useful information between client and servers.

This coarse-grain supervision approach was applied to a lattice calculation involving many independent branch points. Lattice calculations is not the only field that can benefit from coarse-grain supervision. Fuel management calculations is another noticeable field in which numerous full-core calculations are required to test different loading patterns. Again, most of these calculations are independent and can be

dispatched over many processors using the techniques presented in this paper.

ACKNOWLEDGEMENTS

This work was supported by a grant from the Natural Science and Engineering Research Council of Canada.

REFERENCES

1. See `http://www.polymtl.ca/jargon`.
2. G. Marleau, A. Hébert and R. Roy, "New Computational Methods Used in the Lattice Code Dragon," *Proc. Int. Top. Mtg. on Advances in Reactor Physics*, Charleston, USA, March 8-11, 1992.
3. P. Niemeyer and J. Knudsen, "Learning Java," O'Reilly & Associates, 2002. See also `http://www.beanshell.org/`.
4. R. Roy, "The CLE-2000 Tool-Box", Report IGE-163, Institut de Génie Nucléaire, École Polytechnique de Montréal (1999).
5. A. Hébert, G. Marleau and R. Roy, "A Description of the DRAGON Data Structures," Report IGE-232, École Polytechnique de Montréal, December 1997.
6. A. Hébert and R. Roy, "A Programmer's Guide for the GAN Generalized Driver, FORTRAN-77 version", Report IGE-158, Institut de Génie Nucléaire, École Polytechnique de Montréal, December 1994.
7. J. Zukowski, "Java Collections", Apress, Berkeley, 2001.