Java-based coupling for parallel predictive-adaptive domain decomposition

Cécile Germain-Renaud * and Vincent Néri
LRI CNRS – Université Paris 11, Bat 490, Université Paris-Sud, F-91405 Orsay Cedex, France
Tel.: +33 1 69 15 76 55; Fax: +33 1 69 15 65 86;
E-mail: cecile.germain@lri.fr

Adaptive domain decomposition exemplifies the problem of integrating heterogeneous software components with intermediate coupling granularity. This paper describes an experiment where a data-parallel (HPF) client interfaces with a sequential computation server through Java. We show that seamless integration of data-parallelism is possible, but requires most of the tools from the Java palette: Java Native Interface (JNI), Remote Method Invocation (RMI), callbacks and threads.

1. Introduction

Presently or in the near future, high performance computing applications are bound to be built as distributed systems based on an object component architecture [4,7,9]. Some of these components may be written in data-parallel languages and run either on traditional Distributed Memory Architectures or on fast Networks of Workstations (NOWs). The question of integrating such data-parallel components has been raised in multiple contexts. Java-based integration has been discussed by the Parallel Compiler Runtime Consortium [14], and implemented in the Converse framework [3]. CORBA based integration asks for describing data distribution at the IDL level, and defining efficient communication mechanisms [12].

The PPADD (Parallel Predictive-Adaptive Domain Decomposition) project is a large-scale application that naturally requires a distributed implementation on heterogeneous parallel machines. The general application field is solving PDEs through adaptive domain decomposition methods. The entry point to distributed computing is that the main computation (the solver) can see in advance that it will require some adaptation of its mesh structure, and is able to delegate these costly computations before the new data is actually required. Moreover, the solver naturally fits into the data-parallel model, while the mesh computing service is more efficiently performed on shared memory architectures.

In the development of the PPADD project, we were presented with the problem of designing both the solver and the mesh service for efficiency and reusability, *and* to find a smooth way of connecting the components through a LAN.

A good efficiency/reusability tradeoff for the solver was considered to be High Performance Fortran (HPF). The mesh server is written in Fortran 90. Other components, especially a visualization module, will be added later, and will use commercial tools.

We choose to implement the coupling through Java, using the language and the various standard APIs. Smooth coupling precluded using low-level message-passing interfaces. Ideal smooth coupling would require to be able to see HPF and Fortran 90 code as object components, and to invoke remote methods from the HPF objects on the F90 objects. Java Native Interface (JNI) API allows wrapping Fortran 90 (in the limits of C-F90 interoperability, which suffer some severe restrictions), and coupling F77 with Java has been extensively studied [5,9]. The Java Remote Method Invocation (RMI) mechanism allows interoperation with remote Java objects. On the other hand, no standard has yet emerged for wrapping data-parallel objects. So we decided to experiment with the possibility of adapting both the Java JNI and RMI to HPF and to the requirements of our application. This paper presents a preliminary report of this experiment.

The second section describes the application, and motivates the choice of a decoupled architecture. The third section gives an overview of the architecture and points out the requirements for efficiency. The following section describes the Java based implementation of component coupling. Although thoroughly tested for functionality, the architecture has not yet been tested for performance, so the next section will be the conclusion.

*Corresponding author.

Scientific Programming 7 (1999) 185–189
ISSN 1058-9244 / $8.00

2. Application overview

The PPADD project is based on a finite element multilevel scheme for the solution of non-linear PDEs. The terms describing the non-linear interactions between the fine and coarse level are costly to compute, and can be frozen during a time that can be estimated from the behavior of the solution. This multilevel scheme is the mathematical basis for decoupling the solver from the mesh refinement. Moreover, the solution of each approximate non-linear equation uses domain decomposition methods based on dual Schur's complement.

The global architecture is composed of three main modules: the *Solver*, the *Domain Manager* and the *Visualization Interface*. Currently, only the Domain Manager and one version of the Solver are implemented. Besides visualization facilities, future versions will allow the configuration of the Solver for multiple numerical solvers.

2.1. The Solver

At the application level, the Solver is responsible for computing an approximate solution of the initial equation. The solver iterates over time. At each iteration, the behavior of the solution is checked against a too steep variation from from the previous iteration (or from an initial guess at cold boot), as described in [10]. When such a variation occurs, the checking procedure also computes a tolerance delay, and issues a request for mesh refinement. The tolerance delay is the number of non-linear iterations during which the current mesh can remain in use. Moreover, the checking procedure computes two types of indicators: the first one relates the mesh elements to the quality of the solution, and the second one provides information about computing load on a per-element basis.

When a request for mesh refinement has been issued, the computation continues until the tolerance delay is exhausted. At this point, the new mesh is acquired and the time iterations can go ahead.

Currently, mesh refinements are serialized. For a given iteration, only one domain can request refinement, and the request must complete before a new one can be issued. Parallelizing mesh refinement requests does not make sense, because the remesh logic is inherently serial. Pipelining is an interesting possibility.

2.2. The Domain Manager

The Domain Manager acts as a repository for the current state and the history of the finite element mesh and its associated decompositions. Moreover, it provides a unified mesh refinement/domain decomposition service. Finally, it provides the initial mesh at cold boot or on restart. To save memory space, as the computation is not supposed to be out of core, the Domain Manager only sees a simplified version of the mesh data (basic domain structure in the following).

On a mesh refinement request, the Domain Manager always performs the required refinement, and can also decide to split the requesting domain in two subdomains or to merge two subdomains. The decision to split or merge depends on two factors: memory space, and load balancing. Memory space limitation is easy to compute, because the actual memory size needed by the vectors and sparse matrices defined on a domain can be accurately predicted from the basic do-

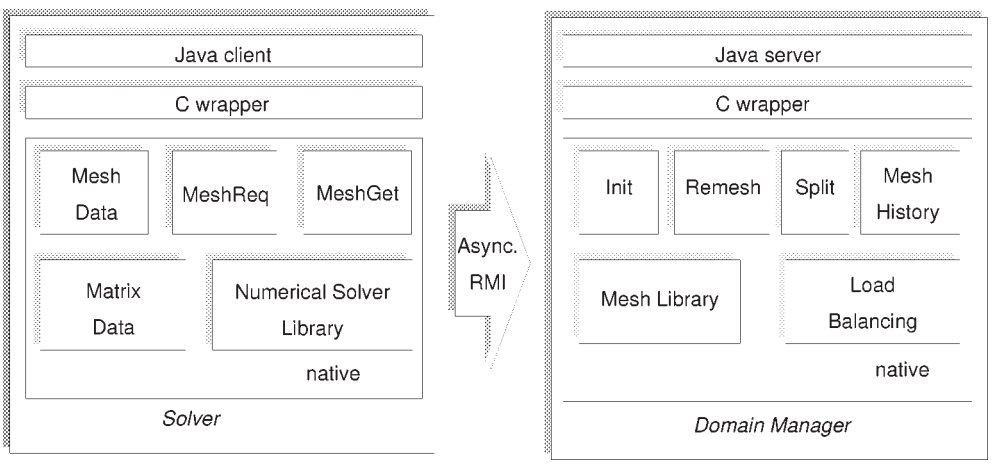

Fig. 1. Architecture overview.

main structure. Load balancing is much trickier, because the convergence time for the non-linear iterations has to be predicted.

2.3. The programming models

The Solver is a data-parallel program, written in High Performance Fortran. It is based on an ongoing effort to develop an HPF library for domain decomposition based sparse computations, which is in turn an adaptation of a Fortran 90 library developed by the Numerical Analysis Laboratory of Paris 11. A discussion of the techniques for handling unstructured computations in HPF is outside the scope of this paper; a survey can be found in [6] and examples for sparse matrix computations in [8] and [17].

The Domain Manager cannot be easily parallelized on a distributed memory machine, because it spends most of its execution time in searching complex data structures. In any case, its interface to the external world is assumed to be a sequential front-end.

Neither Fortran 90, nor HPF as a superset of Fortran 90, are object oriented languages. However, using their modularity and polymorphism features [13], makes it rather easy to interface them with a truly object language.

3. Architecture overview

The application naturally presents a client/server structure. The Solver acts as a client for a service of mesh refinement and domain decomposition provided by the Domain Manager. Here, the most important point is the fact that the computation of a new mesh and the progress in computing a solution based on the old mesh can overlap, at least for a certain time, which is the tolerance delay. Thus, delegating the computation of the new mesh makes sense. As seen before, it is also natural that the two programs run of different machines, because they require different architectures.

The Solver and the Domain Manager are coordinated through Java-based interfaces. These interfaces have two functions: the native/Java wrapper and the mechanism for accessing remote objects. Fig. 1 describes what would be an ideal architecture. On the left hand side Java objects wrap the HPF data and/or code for the Solver; on the right hand side Java objects wrap the Fortran 90 data and/or code for the Domain Manager; on both sides, at least some data can be accessed by both Java and the native codes and each can hold its own data. Finally, method invocation from the Solver to the Domain Manager is asynchronous: invoking the mesh refinement tool should not block the caller.

Actually, only some of the ideal feature are directly available. A very important one is memory management. Simply put, complex dynamic Fortran variables can be allocated and operated on (as Objects) in a Java environment. Hence wrapping true Fortran 90 code is possible, except for the numerous restrictions of C/F90 interoperability, even for the very simple data structure of an array section.

No JNI exists for HPF. Various proposals are ongoing to define how a data-parallel object should be wrapped, either as a mere Object or with more information about its distribution properties. This would clearly be mandatory to have an authentic HPF JNI. However, we observed that the most important need in the PPADD project was a Java-based mechanism to escape from data-parallelism. This takes place when the Solver requests a new mesh. The semantics of this call is strongly asymmetric: currently one, and later several domains, actively call for remeshing, while other domains may be created as a result of the remeshing process. The semantics of HPF_LOCAL answers this need. The semantics of this call also is strongly asynchronous: first, the Solver computation should overlap with the Domain Manager computation, but also with the communication of the new mesh data. Thus, the request for a new mesh must be a true asynchronous RPC, with return parameters set by the called service. This is definitely not the semantics of any data-parallel construct, which imply one thread of execution, nor the semantics of the Java RMI, which is synchronous. However, we show below that the functionality can be implemented through the Java multithreading facility.

4. Implementation

4.1. Memory management

Fortran 90 and, at least in the standard, HPF, afford dynamic data structures, product types and the combination of both, which have been efficiently implemented by commercial compilers each in a proprietary way. A well-designed Fortran 90 code will heavily rely on these complex structures and will define “constructors” for each of them with dynamic memory allocation. The finite element mesh is a typical example of this kind of object, on the Solver and on the Domain Manager as well. The domain decomposed sparse ma-

trix is another example on the Solver. To be usable by Fortran code, such objects must be constructed by Fortran code, because of the proprietary implementation details, and remain valid until de-allocated. Although the status of the native data is not clearly documented in the API specification, we never encountered memory management conflicts. Conversely, large Java arrays have to be accessed by native methods, especially the RMI parameters. The JNI provides the notion of pinning a Java array against Garbage Collection, so that the native method can access this array without copying (provided that the layouts are compatible).

4.2. Java/HPF Interface

When running on a parallel machine, the abstract unique thread of control of data-parallelism is actually implemented through multiple parallel processes running on the parallel processors, in the so-called SPMD mode. Each of these processes must be coupled with a Java VM running on the processor to achieve the functionality of a JNI. A straightforward design would involve one more Java VM as a front-end. However, such a design would either lead to unnecessary communications if the front-end is run on a remote machine, or to load unbalance if the front-end is run on one processor of the parallel machine. Thus no front-end has been implemented, but the Java programs that run in parallel have been designed in SPMD mode so as to offer to the external world the single thread semantics.

A true JNI would have the HPF codes launched by the Java VMs. More precisely, on each processor, the Java VMs would call the sequential code, compiled by an HPF compiler, as native procedures. Unfortunately, executing a parallel procedure is a less standardized process than in the sequential case: for instance, the set-up of the communication layer must complete before executing the actual user code, and there is no standard for all these parallel initializations. Hence, such a design cannot be compiler independent, and needs to know the internal of the compiler. To ensure portability across compilers and machines, we used the Java Invocation API, which allows a Java VM to be loaded into a native application. Here, a Java VM is launched on each processor from inside the HPF code, through HPF_LOCAL subroutines. After this point, the SPMD programs appear as native methods run inside Java threads. However, the SPMD programs are run as a whole. Thus the burden of synchronizing with the Domain Manager is on the HPF program. In practice, we had to mirror in HPF what should be the Java main program.

4.3. Asynchronous RPC

The Solver requests a new mesh through a HPF_LOCAL routine. This routine calls back to the local Java VM. The called-back method forks the execution thread of the requesting processor. One thread continues executing the Solver code, while the other one actually performs a synchronous RMI to the Java server of the Domain Manager.

More precisely, the processor which owns the requesting domain invokes a method that calls the Fortran 90 remesh library (active method). The other processors invoke a method whose role is to synchronize with the active method and to get back a new domain if the Domain Manager decides to split the requesting domain. As the RMI API guarantees that methods invoked from different physical machines execute on different threads, this synchronization uses standard Java thread primitives.

Currently, the RMIs return a Java Object which encapsulates all the arrays required to describe the modified domain and the new one if the requesting domain has been split or merged. It has been shown that moving Objects through RMI is much less efficient than moving primitive arrays, because of the cost of marshaling/unmarshaling Objects instead of primitive arrays [2]. Designing a domain linearization layer to optimize this transfer could be realized if necessary.

Eventually, the thread associated with the Solver code reaches the point where it needs the new mesh. Then, the finalizer of the RPC is called, as another HPF_LOCAL routine. This routine joins with the RMI thread, hence blocks if it happens that the RMI has not completed yet. Then the mesh data are updated with the return parameters. This is the only non-overlapped copy-back overhead, which is unavoidable because all the data related to the old mesh (matrices and solution vectors) are in use up to this point in time.

5. Conclusion

Although we cannot present performance results, we are confident that the architecture will be as efficient as the underlying software allows for. Preliminary evaluations of the non-linear solver execution time indicate that it it will largely overlap the remesh time. Optimizations are possible for best exploitation of the RMI mechanism; we checked that on all the machines which run the Solver (IBM SP2 and Linux PCs), and the Do-

main manager (Sun workstations), the Java GC is actually able to pin down arrays.

Besides completing this experiment, future work will try to generalize this experiment to a more general HPF JNI. We plan to use an open compiler infrastructure such as ADAPTOR [1] to get rid of the technical problem of compiler-dependent initialization problems. Such an interface will probably have to use some of the Java extensions designed for integrating data and task parallelism [16,15,11] in order to be able to express the semantics of asynchronous RPC as described in this paper.

Acknowledgments

The experiments were performed on the SP-2 from the Computing Resource Center (CRI) of Paris-11 and the National Computing Center (CNUSC).

References

[1] T. Brandes and F. Zimmermann, ADAPTOR: A transformation tool for HPF programs, in: *Programming Environments for Massively Distributed Systems*, 1994.

[2] F. Berg et al., Java RMI performance and object model interoperability: Experiments with Java/HPC++, in: *Java98 ACM Workshop*, 1998.

[3] L.V. Kale, M. Bhandarkar and T. Wilmarth, Design and implementation of parallel Java with global object space, in: *Proc. Int. Conf. Parallel and Distributed Processing*, July 1997.

[4] H. Casanova and J. Dongarra, Netsolve: A network server for solving computational science problems, in: *Proc. Supercomputing '96*.

[5] H. Casanova, J. Dongarra and D.M. Doolin, Java access to numerical libraries, in: *ACM Workshop on Java for Science and Engineering Computations*, June 97. *Concurrency: Practice and Experience*, 1997.

[6] F. Coelho, C. Germain and J.-L. Pazat, Compiling HPF, in: *The Data Parallel Programming Model*, 1996. Lecture Notes Comput. Sci., Vol. 1132.

[7] G. Fox and W. Furmanski, Towards Web/Java based high performance distributed computing, in: *Proc. 5th IEE Int. Symp. on High Performance Distributed Computing*, 1996.

[8] C. Germain, J. Laminie, M. Pallud and D. Etiemble. An HPF case study of a domain-decomposition based irregular application, in: *PaCT '97*, Lecture Notes Comput. Sci., Vol. 1277.

[9] V. Getov, S. Flynn-Hummel and S. Mintchev, High-performance parallel programming in Java: Exploiting native libraries, in: *Java98 ACM Workshop*, 1998, http://www.cs.ucsb.edu/conferences/java98/

[10] C. Calgaro, J. Laminie and M. Temam, Dynamic multilevel schemes for the solution of evolution equations by finite elements discretization, *Applied Numerical Maths*, 1997.

[11] P. Launay and J.-L. Pazat, A framework for parallel programming in Java, in: *HPCN '98*, Lecture Notes Comput. Sci., April 1998.

[12] K. Keahey and D. Gannon, PARDIS: A parallel approach to CORBA, in: *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing* (best paper award), August 1997.

[13] L. Machiels and M.O. Deville, Fortran 90: An entry to object-oriented programming for the solution of partial differential equations, *ACM Trans. Mathematical Software*, March 1997.

[14] *HPCC and Java – A preliminary report by the Parallel Compiler Runtime Consortium*, http://www.npac.syr.edu/users/gcf/hpjava.html

[15] M. Philippsen and M. Zenger, JavaParty – transparent remote objects in Java, in: *PPoPP*, June 1997.

[16] K. van Reeuwijk, A.J.C. van Gemund and H.J. Sips, Spar: A programming language for semi-automatic compilation of parallel programs, *Concurrency: Practice and Experience*, Nov. 1997.

[17] E. Sturler and D. Loher, Parallel iterative solvers for irregular sparse matrices in High Performance Fortran, *Journal of Future Generation Computer Systems*, 1998.