JAVAGRID: AN INNOVATIVE SOFTWARE FOR HPCC A PAPER FOR ECCOMAS COMPUTATIONAL FLUID DYNAMICS CONFERENCE, SWANSEA 2001

Jochem Hauser[*], Thorsten Ludewig[*], Torsten Gollnick[*], and Roy D. Williams[┬]

[*]Dept. of High Performance Computing
Center of Logistics and Expert Systems (CLE) GmbH
Salzgitter, Germany
Email: info@cle.de, web page: http://www.cle.de/cfd/

[┬]Center of Advanced Computational Research
California Institute of Technology
Pasadena, U.S.A.
web page: http://www.cacr.caltech.edu/

Key words: Java HPC, client-server computation, OOP, Internet-based computing,Internet-based data access, diverse scientific and engineering disciplines, collaborative engineering, portable HPC and geometry framework, legacy code integration, architecture independence, HPC without libraries, complex 3D geometries , just in time solver, remote visualization and X3D.

Abstract.

In this paper we describe the *JavaGrid* concept that underlies the software developed for high performance computing and communication in science and engineering. *JavaGrid* provides a package for parallelization based on Java threads, a geometry package for handling 2D and 3D structured as well as unstructured grids, a generic solver and a solver template to model a system of integral conservation laws. *JavaGrid* provides both client and server software and allows to send a specific solver at run time from the client to the server, overriding the server's default solver. For instance, this might be a computational fluid dynamics solver, while the client wishes to execute an electrodynamics solver. However, both solvers could be based on the template solver provided. Setting up a new solver is a straightforward process, since only the physics equations have to be implemented for a single subdomain. Geometry handling, parallelzation (i.e. updating the boundary of neighboring subdomains) and communication is handled by *JavaGrid*. It is also possible to incorporate so called legacy solvers, written in other languages. A Virtual Visualization toolkit for remote visualization is also provided. The paper describes the current status of the *JavaGrid* project and presents performance figures.

1 DISCIPLINE: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS ON THE INTERNET

This is an age of possibility, and information technology is the driving force behind this change that occurs on a global range. *High Performance Computing and Communications (HPCC)* is one of the key technologies of the new economy, [1]. The bandwidth of the Internet will increase rapidly over the next five years, and a communication speed of 1 Gbit/s should be available in 2005, perhaps even earlier. Software that makes *HPC* possible on the Internet is the enabling technology for computer simulation in many areas of science and engineering.

The need for accurate three-dimensional simulation in numerous areas of computationally intensive industrial applications as well as in many fields of engineering and science, including the rapidly evolving field of bioscience, requires the development of ever more powerful *HPCC* resources for a computational Grid based on the Internet.

During the last five years there has been enormous change in computing and communications hardware. In the midst of these demands and changes the question arises how to build the simulation *software* capable of exploiting the new hardware, dealing with complex three-dimensional geometries, running in parallel, being platform (architecture) independent, and being able to access geographically distributed computational resources via the Internet. In addition, the questions of geometric modeling of complex configurations (preprocessing stage) and visualization of computed results arise (post-processing). Visualization and solution feature extraction along with data extraction and compression are of prime importance to deliver the relevant information to the design engineer.

To satisfy the above demands along with the additional requirements of code parallelism, code maintainability and portability, code security, graphics user interfaces (GUI), and data base connectivity to visualize or access data that is distributed over different computer architectures connected by the Web, requires a completely new approach. With procedural programming languages like Fortran or C or even C++, these goals cannot efficiently be achieved.

Attempts have been made to provide such a computational Grid, see [2, 3], by developing a special computational infrastructure, providing both services and programming tools. With the advent of the Java language in 1996, a general object-oriented programming tool is available that provides full coverage of all programming needs on the Internet and also ensures security. Thus the computational Grid for the Internet can be built entirely in Java in a transparent, object-based approach, termed ***JavaGrid***. This includes high-performance (parallel) computing [4-14] as well as data intensive computing utilizing the available computing platforms and network infrastructure.

JavaGrid also provides the services for complex geometry handling - applications in science and engineering very often require dealing with complex 3D geometries. Visualization of data is managed by a *Virtual Visualization Toolkit*, based on *Java3D*.

The construction of *JavaGrid* provides full portability to any computing platform, and, at the same time, has the infrastructure in place to couple so called legacy solvers from other languages through *RMI over IIOP, CORBA* or JNI [5] . In addition, all necessary layers for geometry handling, parallelization, scheduling, Internet connectivity, and post processing in form of the Virtual Visualization Toolkit (VVT) are provided in the framework of *JavaGrid*. Hence, a *solver* only needs to contain the physics and the numerics of the simulation task for a *single block or a single domain* (subdomain). In other words, such a solver does not need to know anything about the geometry data or the parallelization, and thus has a very simple structure. It can be tested independently before its integration.

2 SCOPE OF *JAVAGRID*

JavaGrid is a revolutionary computing software that dramatically improves the ability to quickly create new kinds of software systems across the whole field of science and engineering embedded in a Web-based environment. ***JavaGrid*** is a software platform and virtual computing environment that enables scientific and engineering computation of large-scale problems in a Web-based computational grid environment, integrating computer resources at different, geographically distributed, sites. ***JavaGrid*** enables the user to create his simulation software at the client site at run time by using a Java based browser GUI. The solver package composed by this GUI is sent in binary form to the server site, replacing the default simulation solver package.

JavaGrid is a completely Java based software environment for the the user/ developer of *HPC* software. ***JavaGrid*** takes care of the difficult tasks of handling very complex geometries (aircraft, spacecraft, biological cells, semiconductor devices, turbines, cars, ships etc.) and the parallelization of the simulation code as well as its implementation on the Internet. ***JavaGrid*** builds the computational Grid, and provides both the geometry layer and parallel layer as well as an interface to attach any arbitrary solver package to it. ***JavaGrid*** is implemented on the client site, where the user resides, and on the compute server where the computations are to be performed. It also can access one or more data servers, distributed over the Internet. A default solver package resides on the server site. For instance, this may be a fluid dynamics solver. If the client decides that it will use this solver, only the necessary data has to be collected and sent to the server. In case a totally different solver is needed, e.g., a solver for Maxwell's equations to compute, for instance, the electromagnetic signature of a ship or aircraft or to simulate the trajectories of an ionized plasma beam of an ion thruster, the correct solver object has to be sent from the client to the server at run time. As described above, the new solver is created through the GUI at the client site at run time. This solver object is sent in binary form to ensure code security. If the solver object is written in Java, the Remote Method Invocation *(RMI)* class is used, if not, the Common Request Broker Architecture (*CORBA*) or the Java Native Interface (JNI) is employed to integrate so called legacy solvers. The server does not need to know anything about the solver as long as the solver interface is correctly implemented and any kind of simulation application is supported. The parallelization is entirely based on the Java *thread* concept (see next chapter for details). This thread concept has substantial advantages over the *PVM* or *MPI* library parallelization approach, since it is part of the Java language. Hence, no additional parallelization libraries are needed.

JavaGrid also provides a third layer, the solver package layer, to be implemented on the *client*. This layer is a Java interface, that is, it contains all methods (functions in the context of a procedural language) to construct a solver whose physics is governed by a set of conservation laws. An interface in the Java sense provides the overall structure, but does not actually implement the method bodies, i.e., the numerical schemes and the number and type of physical equations. This *JavaSolverInterface* therefore provides the software infrastructure to the the other two layers, and thus is usable for a large class of computational problems. It is well known that the Navier-Stokes equations (fluid dynamics), Maxwell's equations (electromagnetics, including semiconductor simulation) as well as Schrödinger's equation (quantum mechanics) can be cast in such a form. Thus, a large class of solvers can be directly derived from this concept. The usage of this solver package, however, is not mandatory, and any solver can be sent by the *client* at run time. All solvers extend the *generic solver* class, and in case a solver does not need to deal with geometry, the generic solver class is used directly, instead of the conservation law solver class.

JavaGrid provides the coupling to any existing solver, but freeing this solver from all the unnecessary burden of providing its own geometrical and parallel computational infrastructure.

Because of Java's unique features, ***JavaGrid*** is completely portable, and can be used on any computer architecture across the Internet.

3 OBJECTIVES OF JAVAGRID

In the following we will outline the ***JavaGrid*** objectives. ***JavaGrid*** promises to provide the combined power of networked computational resources for solving most complex scientific and engineering problems both in geometry and in physics. The grid comprises clients, a server (*SMP* parallel architecture), and data servers.

The concept of *Java Spaces* allows the extension to distributed compute servers or a farm of compute servers, but this would come on top of the current ***JavaGrid*** and is not pursued in this proposal.

Clients are used to communicate the solver classes in binary form to the server at run time replacing the default solver located on the server, that is, the kind of solver being used is only determined at run time. In addition, clients are used for steering or navigating the simulation application as well as for visualization. *Clients* have their own identity, that is, once this identity has been established, any computer on the Internet can be used to run this client process.

Java is the language of choice for High Performance Computing and Communications because of its unique features with its built in *threads* for parallelization and its highly performant and efficient *socket* programming as well as *Remote Method Invocation* (*RMI*) [5] to facilitate communication. ***JavaGrid*** will comprise computational resources connected by the Internet to create a universal source of computing power, forming a computational grid.

The complexity of the computational grid is hidden from the user and/or the developer that is, no knowledge of the underlying infrastructure is needed. Java packages are available that provide the handling of complex three-dimensional geometries both for structured as well as for unstructured grids, a parallel framework for dynamic loadbalancing, a framework for the set up and the numerical solution of the governing physical equations, a graphics user interface for collecting input along with the framework to obtain the geometry data that might reside on geographically distributed computers. and a visualization package based on the *Java3D* standard.

Thus, the developer can concentrate on the scientific components of his problem focusing on the equations that describe the physics. ***JavaGrid*** also provides a library to computing the solutions of physical systems that are expressed as a system of hyperbolic conservation laws, meaning that each equation of the system corresponds to a physical quantity that is generally conserved. For example, the system may be governed by the conservation of energy, momentum, and mass. Mass conservation may be extended to conservation of individual molecular or atomic species rather than just total mass.

We envision a layered architecture, where each package is implemented in terms of the packages below. There will be a GUI package, which instantiates objects from the *physics-numerics* package, which is implemented with the *solver* package, which is implemented in turn by the *parallel* and *geometry* (structured multi-block and unstructured grids) packages.

Since the ***JavaGrid*** strategy is based on the concept of functional layers, the *solver* layer could be omitted, using the *geometry* and *parallel* layers only. In this way, a different system of physical equations could be implemented, interfacing a new physics-numerics package to the geometry and parallel packages. The *JavaGrid* could also be reduced to the *geometry* package if the developer decides to interface his own parallel *package*.

4 SPECIAL APPLICATION OF JAVAGRID

In this paper, for details see below, a special application is foreseen that serves a large number of simulations application in many fields of science and engineering. *JavaGrid* provides a special framework for these applications, substantially facilitating the development of new simulation software in different areas, simply by extending existing classes.

It should be noted that a very wide class of scientific and engineering problems is covered by the current *JavaGrid* approach, ranging from quantum mechanics, internal and external compressible as well as incompressible flows, including chemically reacting flows, and radiation as well as turbulent flows. In addition, Maxwell's equations or the magneto-hydrodynamics equations can be cast in this form, too. Many problems in the rapidly evolving field of bioscience fall under this category, too.

All these problems can be described by the general case of a nonlinear system of hyperbolic conservation laws. Diffusion processes can be included as well. Since hyperbolic laws are marked by a finite propagation speed, fluxes have to be calculated. The physical interpretation of these fluxes depends on the problem to be simulated. The fundamental structure of the simulation model, however, remains unchanged. Fluxes can always be partitioned in their hyperbolic (finite propagation speed) part and other processes like diffusion, dispersion etc. A transformation will be used from physical space to computational space that comprises a set of connected blocks (regular shaped boxes for structured grids) or a set of connected domains (equal size, unstructured grid). The boundaries of neighboring blocks or domains are connected by a set of halo cells (very often two halo cells are used, i.e., there is an overlap of two cells between any two neighboring blocks or domains), but this number can be specified at run time.

5 STRUCTURE OF JAVAGRID

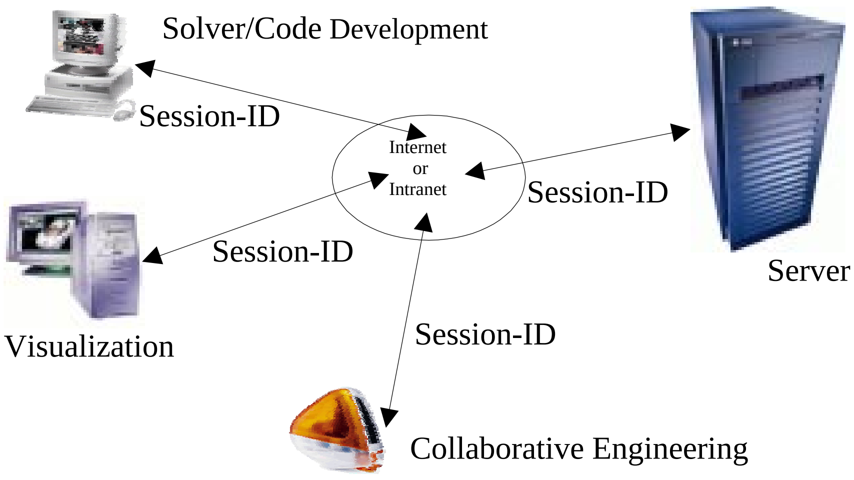

Figure 1 JavaGrid: Architecture Overview

The objective of this work is to develop, demonstrate and verify advanced object-oriented Java packages both for the *client* and *server* sites as part of the on-going, long-term IT research program, addressing a wide variety of HPCC issues. Applications include handling of arbitrary complex three-dimensional geometries, a general, solver independent, parallelization layer based on the Java *thread* concept providing automated *static* and *dynamic load* balancing, a Java template *generic solver* based on the integral conservation law approach, a Java wrapper class solver for integrating solvers (legacy code) written in a different programming language, and a Java based compressible flow solver including nonequlibrium gas dynamics for high speed flows as well as post processing visualization and steering software for *client-server* interaction.

The *JavaGrid* software comprises ten major packages, described below. It should be noted that some software packages are installed on the client side only, while others are installed on the server side, and some packages are shared. Multiple sessions are possible, i.e., the server can communicate with more than one client at a time. Client and server architectures need to be connected via the Internet. Input data may be retrieved from a file or a URL and can be located at any of the clients, the server, or somewhere on the Web.

5.1 GUI Browser and Graphics User Interface

5.1.1 Solver GUI Browser

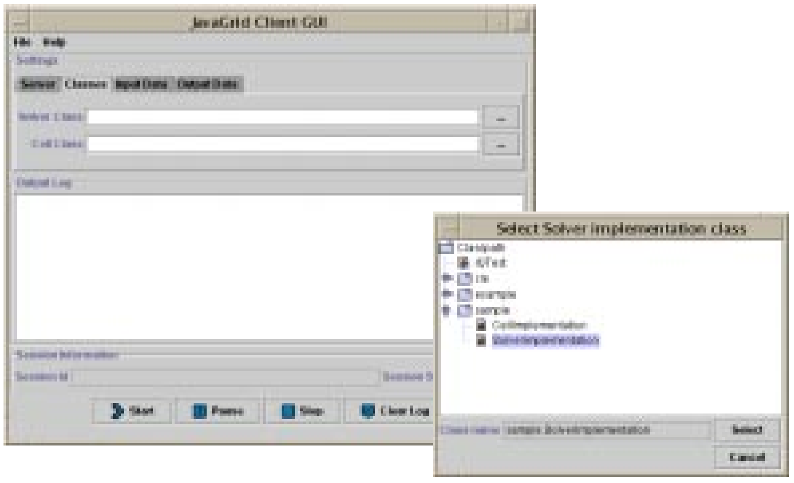

Figure 2   This picture shows the current status of JavaGrid: Client Graphical User Interface (GUI) with an opened class browser dialog for selecting the solver class to be used.

This browser GUI lets the user interactively create his application software at the client site. The solver GUI browser will interact with the solver construction process in that it lets the user select the *solver* and *cell* classes (finite volume if needed) to be added to the actual *client* package.

In that way, the ability to create the proper simulation software package at run time at the *client* site is accomplished. This solver class will be added to the client package and then be sent through the Web to the *server* site, replacing the *server default solver*. Thus, it is possible to dynamically create solver packages for many different applications that automatically have access to the geometry package and automatically run in parallel on the target architecture, without the usage of a parallelization library and without any parallelization effort for the solver software. Complete data security is maintained through Java's built in security mechanism.

5.1.2 Graphics User Interface

This GUI provides the interface between the user and the *JavaGrid* collecting the information to run the parallel application. On the other hand, the GUI also provides guidelines for the user to facilitate the usage of the application. The user starts a session and obtains a session ID that subsequently can be used to access the server from any other machine connected to the computational grid anywhere on the Internet.

5.2 Geometry Package

The geometry package allows the handling of arbitrarily complex grids. The solver package only needs to know a single domain, i.e., the geometry handling and the parallelization is taken care of by these two *JavaGrid* layers. A solver package can be in Java or in any other language. If the solver is written in Java, the *RMI* methods implemented in the parallelization layer will provide the communication for the computational grid. Any other language is interfaced to the *JavaGrid* environment through a wrapper class and *CORBA*.

5.2.1 Structured Grid Domains

Any structured multiblock grid written in NASA standard format *Plot3D* will be supported. The connectivity information between subdomains or blocks is automatically reconstructed form the grid point coordinates.

5.2.2 Unstructured Grid Domains

Unstructured hexahedra grids are supported, for instance, the *Nastran* or *StarCD* format is supported. The domain decomposition is done internally, based on a recursive bisection algorithm as described in [26]. It is foreseen to also support other element types, such as tetrahedra, prisms, or pyramids since only the number of faces and the computation of the volume need to be changed, the overall class structure of a cell remains the same.

5.2.3 Metric Computation

JavaGrid expects the grid point coordinates either in form of a set of blocks (NASA Plot3D format) or in form of an unstructured grid in Nastran or StarCD format. For a multiblock grid, block connectivity is reconstructed from grid point coordinates. The incorporation of other data formats is relatively easy and filters can be integrated in the input class. For each cell the complete metric information is computed, i.e., its midpoint coordinates, volume, face values, and normal vectors. It is assumed that the conservation laws are solved in computational space and therefore all necessary first and second derivatives are numerically computed and made available to the solver class. Thus the solver does not need to know about geometry.

5.3 JavaGrid Parallelization Package

Multithreading is the way in *JavaGrid* to get improved performance from a code, because all modern machines have extra processors that can run the extra *threads*. But in C is it much more difficult to manage threads than it is in Java therefore programmers simply do not use threads very much. But in Java, it is very easy to spawn a new thread, and therefore use of threads is much more natural and widespread.

5.3.1 JavaGrid Execution Strategy

An architecture independent execution strategy for *JavaGrid* is implemented in this package. In order to execute the *JavaGrid* along with the specified solver code, a series of events on both the *client* and the *server*, as described below, is needed. In this package, these stages are implemented in form of three packages, denoted as *client*, *server*, and *share*.

[**Server: compile server programs**] Compile (javac) the Java files (extension .java) of the server module on the server.

[Client: compile client programs] Compile the Java files on the client.

[Server: generate stub and skeleton codes using rmic compiler] Generate the stub (client) and skeleton (server) code by running the rmi compiler (*rmic*) on the server and copy the stub code to the client. The stub code contains the signatures of the remote methods and provides the necessary information for the client code.

[**Server: registry setup**] Start the *registry* to enlist all remote objects on the server. A server object is registered by giving a reference and a name (unique string) to the *registry*. On the client the *Naming.lookup()* method of the stub code accesses the remote object on the server by giving the server name in *URL* format, combined with the name of the server object, as has been registered in the *registry*.

[**Server: object registration (binding)**] Start the code that registers (*binding*) all objects of class implementation on the server, i.e. the *JpMaster* process.

[**Client: remote object lookup**] Start a program that looks up the registered remote server objects. The *JpClient* can then manipulate these remote objects by invoking methods, and create new remote objects.

5.3.2 Client Package

In this package the general structure of the *client* package is developed. The approach is based on the *client-server* concept and thus allows to perform a parallel computation using the Internet. The *stub* class, *s*ee next Section, along with the *client* code resides on the *client* computer, and the *skeleton* class is on the *server* machine. The communication between *client* and *server* takes place through these two objects. In the last step, the code on the *client* is started using the same *rmi* address, see next Section, Since both *client* and *server* know the *shared interface* code that contains an *interface* for the *JPSolver* class, implemented on the *client* site, the *client* **can send its own solver object at run time** to replace the default solver on the *server* site.

5.3.3 Server Package

In this package the general structure of the *server* package is implemented to establish communication on the *server* side. To this end, the compiler for *Remote Method Invocation* objects, *rmic*, - the *rmic* is part of the Java Development Kit (JDK) - has to be evoked to produce the *stub* and *skeleton* classes, needed for communication between *client* and *server*. The *stub* classes, then, are implemented on the *client*. Next, the so called *rmiregistry* is started on the *server* to register all objects that can perform communication. The *rmiregistry* command is used for this purpose. In general, the registry now is ready to communicate over port 1080 and listens to communication requests. In the next stage, the *server* code is started and the objects for communication are actually registered. When the *server* is started an address is supplied in form of an *rmi* address, i.e. *rmi://hostname/RMIObject* where *hostname* is the name of the *server*. The *RMIObject* name can be any name, but it must be the same name for both *client* and *server*. A domain name service (DNS) must be enabled that translates this name into a valid internet protocol (IP) address. The *server* knowing the interface of the *JPSolver* object, has the necessary information about the signature of all solver methods (in non-object oriented terminology methods are referred to as functions) and thus knows how to handle the solver object. The *JavaGrid* parallel framework does not know anything about the numerics or physics implemented in a solver object. It provides, however, the necessary parallel infrastructure for all solver objects that implement the *JPSolver* interface. Hence, parallelization is done once and for all, and very different solver objects can be constructed, resulting in a parallel code that solves problems for a wide range in science and engineering.

5.3.4 Share Package

The shared part is in form of a Java *interface* that has to be implemented by either the client or the server. It contains all Java *interfaces* needed by both the *client* and the *server* to enable the communication over the Internet.

5.3.5 Inter-Domain Communication and Synchronization

For the integral conservation law solver, essentially, each subdomain or block of the computation is alternating between computation and data exchange.

The concept of ghost or halo points, is used to model the overlap between neighboring domains or blocks. These subdomains have to communicate after each iteration step to update their boundary (ghost) points. In this regard, parallelization is simply done by introducing a new (interblock) boundary condition. For a multiblock solver, this condition was already present in the sequential code, because complex geometries had to be modeled using the multiblock concept. It should be noted that the multiblock concept, beside numerical advantages, is not subject to the severe performance reduction caused by frequent cache misses for some modern architectures.

In *JavaGrid*, subdomains or blocks are connected via edges (2D) or faces (3D), but not via vertices. That means, communication takes place only across edges, but not across diagonals. Hence, each block is connected to at most four neighbors in 2D (six neighbors in 3D). If only first derivatives have to be computed numerically, diagonal points are not needed. However, for second derivatives the computational stencil needs these diagonal values. Since communication does not take place across diagonals, these values are not explicitly updated. Therefore, a different computational stencil is implemented that computes the missing value from its neighbors, omitting the diagonal value. The scheme retains the same numerical order, but the truncation error changes. For instance, if viscous terms have to be computed in the Navier-Stokes equations, this practice has shown to be both accurate and effective. In this regard, there is a minor difference between the sequential and the parallel numerical algorithms.

The compute phase consists of computing fluxes at cell boundaries, then adding (subtracting) the incoming (outgoing) flux from the values of the primitive variables in each cell.

5.3.6 Macroscopic and Microscopic Parallelization

Multithreading provides a unique advantage, namely that it allows for both macroscopic and microscopic parallelization, a concept explained below.

Independent on the gridding structure of the solution domain, eventually a set of subdomains is obtained. For an unstuctured grid, a recursive bisection algorithm is used to generate subdomains of equal size with boundaries minimizing communication. For a structured grid, a set of blocks is available from the grid generator. These blocks are of different size, but blocks could be grouped to form subdomains of equal size. The only differenece between structured and unstructured subdomains is the way the individual cells (finite volume) have to be accessed and their neighboring cells to be found. In Java, the concept of *collection* as a general datastructure is available, providing so called *iterators* to access individual elements. Thus, the differences between unstructured or structured grids are marginal, existing only at the level of the *iterator*. Therefore, each subdomain is run within its own thread, completely independent of all other threads. The mapping of threads onto the set of processors as well as thread scheduling is entirely left to the underlying operating system. The issues of static and of dynamic loadbalancing are not the concern of JavaGrid.

Within the thread of a subdomain, additional threads can be started to parallelize the numerical algorithm itself. A strategy will be worked out for both parallel scaling and optimizing parallel speedup. On the one hand, a larger thread pool leads to a better usage of the processors, on the other hand, thread administration causes overhead. In Java, thread creation is straightforward and tens of thousands of threads are possible. In principle, a thread for each cell could be started. Architectures like the Tera machine, support the thread concept by dedicated hardware, for instance, each processor of the Tera machine runs 128 hardware threads.

With the thread concept as parallelization, there is an unlimited number of options to speed up parallelization, since parallel fine tuning by threads on all levels of a computation is feasible.

5.4 JavaGrid Generic Solver Package

This package provides a *generic solver* package from which more specific solvers, such as a solver based on conservation law formulation can be derived.

5.4.1 Generic Solver Class

There is a *generic solver* class, *JpSolver*, whose task is to provide the general structure of a solver, simplifying the programming of a new solver or allowing the "modernization" of a legacy code by wrapping it inside the appropriate structure. A solver class for a structured multiblock grid (here a grid is a collection of *grid points* that describe the solution domain) or an unstructured grid can be extended from this *generic solver* class. The numerical solver is based on an integral formulation and thus a description of a *cell* class is needed.

The generic solver class from which the numerical solver class is derived does not know anything about grids or fluxes etc. However, a template implementation (Java *interface*) for a general conservation law solver is provided, making it straightforward to design the proper *fluxes*, *limiter,* and *cell* methods for a specific implementation.

The *generic solver* class has to be used in case a gridless (i.e., no grid points are needed) simulation application (e.g., parallel Mandelbrot computation, see below) was to be implemented.

5.5 Conservation Law Solver Package

Extending the *generic solver* concept one step further is the *template solver* package, implementing a set of integral conservation laws. We deal with a geometrically complex domain G. Conceptually, G is a compact, smooth manifold that is a subset of *dim*-dimensional Euclidean space, where dim is called the dimension of G, and is usually 2 or 3. G may be represented by a manifold of lower dimensionally (e.g. axisymmetric geometry), which we call the representation dimension if it is necessary to distinguish it from the domain dimension.

In the *template solver* package, the classes have physical names, for example, in a flow solver TurbulentEnergy, Flux, Limiter, and so on. Here it is decided whether to use implicit or explicit methods, it is decided if convergence has happened and also if a special submodel should be used. In the *template solver* package, there are implemented various kinds of flux limiters, but it is in the *physics* package that it is decided which one to use. Also, in the *template solver* package are ways to switch on and off certain kinds of activity on a cell-by-cell basis, but it is in the *physics* package that we set the criteria by which this masking happens.

5.6 Java Based Compressible Fluid Dynamics Solver

In this package a compressible flow solver is implemented, filling out the method bodies of the *template solver*. *JparNSS* is a Java based flow solver, following the strategy developed in

[8-12] for the turbulent compressible Navier-Stokes equations that uses the geometry and parallelization layers of the *JavaGrid* environment. This code serves as the large-scale example solver, demonstrating the interfacing of a complex conservation law based solver to *JavaGrid*. The solver resides on the *client* computer and replaces the default solver on the *server* computer at run time, i.e., only the binary solver classes are sent from the client to the server, thus guaranteeing security. Here it is assumed that both the *client* and *server* objects are written in the Java language.

5.7 Integration of Existing Solvers

In general, there exist many different legacy solvers, but in principle all can be wrapped by a Java layer, while the underlying object is written in a different language. Communication must be possible between the Java object and the foreign object, regardless of what language it was originally written in.

5.7.1 CORBA Based Interface to Existing Solvers

In order to achieve this goal, the "common object request broker architecture" or *CORBA* standard, by the Object Management Group or OMG (www.omg.org) defines a common mechanism for interchanging data between clients and server. The Object Request Broker (*ORB*) acts as a kind of universal translator for interobject communication. Objects do not talk directly to each other but use object brokers that are located across a network, communicating with each other, based on a protocol specification termed Inter Object Protocol (IOF).

The Interface Definition Language (IDL) is used to specify the signatures of the messages and the data types that objects can send. IDL have a very similar look and feel compared to Java interfaces. An IDL specification will be applied for an object written in Fortran and C, namely a Fortran/C compressible flow solver. This will serve as a complex demonstration that so called legacy codes can be interfaced with the *JavaGrid* environment, enabling such a solver to inherit the full capability of both the geometry and parallel layers implemented in *JavaGrid*.

5.7.2 JNI Interface to Existing Solvers

The Java Native Interface (JNI) is the native programming interface in Java that is part of the JDK. The JNI allows Java code that runs within a Java Virtual Machine (VM) to cooperate with applications and libraries written in other languages, such as C, C++, FORTRAN, and assembly. By writing programs using the JNI, the *programmer ensures* that the code is completely portable across all platforms. The *JavaGrid* system integrates the legacy codes directly via shared libraries build upon existing solvers.

5.7.2.1 The Invocation API

In addition, the *Invocation API* allows to embed the Java Virtual Machine in native applications. In this case, RMI, the Remote Method Invocation, a pure Java Object Communication API can be used to interact with the *JavaGrid* environment.

5.8 Gridless Application

The usage of the *generic solver* class and the *parallel layer* will be demonstrated by developing a package for the parallel computation of the Mandelbrot set.

This code tests the self-scheduling of threads. An independent iterative calculation takes place at each grid point, where the number of iterations varies greatly from point to point. We partition the solution space into subdomains; where each subdomain is large enough that thread overhead will be much less than the computational work associated with the subdomain, but each subdomain should be small enough that there are many subdomains for each processor.

Although this program is embarrassingly parallel, it exhibits a new feature, namely the computational load depends on the position within the solution domain, which is a rectangle in this case. Dynamic load balancing would be needed to run such an application successfully on a large parallel architecture. Using *PVM* or *MPI*, the user has to provide a sophisticated algorithm to achieve this feature, requiring a lengthy piece of code. Using the Java *thread* concept, *dynamic* load balancing is provided by the operating system.

6 JAVA NUMERICAL PERFORMANCE AND HPC ARCHITECTURE COMPARISON

In this chapter the numerical and the parallel (thread) scalability are investigated. The data presented here can be considered as the continuation of the figures given in [4,5]. Special emphasis is given to the *thread scaling* behavior on the various target architectures.

6.0.1 Numerical Performance and HPC Architecture Comparison

First, a brief discussion of Java code generation and code execution is given. From this discussion, the testing strategy is derived. One of the attractive features of Java is its portability, the idea of “write once, run anywhere”. The portability is obtained because Java is generally not compiled to a platform-specific executable, but converted to so-called *byte-code*, which is in turn executed by a platform-specific interpreter. While these extra layers of insulation provide portability, they may have a performance impact. However, many companies, including Hewlett-Packard, IBM and Sun, offer Java compilers, that create native code directly, and should provide competitive performance with C or Fortran.

Java runs a garbage-collector thread, reclaiming memory that is no longer needed by the application. This is additional overhead which takes away resources from the numerical application, but at the same time it provides an enormous boost to programmer productivity. The programmer is thinking about the physics of the code instead of spending hours chasing mysterious memory leaks that *always* beset large C++ projects.

Although Java programs are statically compiled, there is still a need to do some runtime checking. In particular, null reference checking, array bounds checking, and runtime type checking can't be done at compile time. This makes Java programs more robust, but it also makes the generated code a little slower than the equivalent C program. However, many of these checks can be eliminated at runtime by the native code generator and, for instance, in [6, 7], it is demonstrated that Java, despite the additional services, can indeed match or even exceed the speed of corresponding C or Fortran codes.

JDK Release	*VM type*	*warmup*		*production*	
		time in sec	*MFLOPS*	*time in sec*	*MFLOPS*
jdk 1.1.8	sunwjit	56.086	9.628	55.783	9.680
jdk 1.2.2_07	sunwjit	21.387	25.248	21.368	25.271
jdk 1.3.0_02	hotspot - client	37.896	14.249	37.748	14.305
	hotspot - server	30.226	17.865	19.993	27.009
jdk 1.3.1rc2	hotspot - client	37.066	14.568	37.101	14.554
	hotspot - server	25.162	21.460	9.575	56.396

Table 1 Sequential matrix multiplication for a 30x30 matrix running 10,000 iterations on a 4 CPU (400MHz) Sun Enterprise 450 using different Java Virtual Machines. The warmup phase is needed by the so called hotspot VMs to automatically detect the computational intensive parts of the program. ***Note that the current hotspot server version, jdk1.3.1rc2, (release candidate 2) of the new upcoming release 1.3.1 is more than two times faster as the previous.***

Compared to the performance figures in [4] where **19.31** Mflops were obtained for the 30x30 matrix multiplication utilizing the Java VM version jdk1.2.01 dev 06, the new *jdk1.3.1rc2* provides **56.4** Mflops, which is a **threefold increase in execution speed.**

Currently several versions of *JavaGrid* are being tested generated by different Java compilers, and run on Linux, Sun, HP, IBM, and SGI architectures. The numerical performance of the Java fluid dynamics solver, *JparNSS*, will be measured and compared to a similar code written in C/Fortran. In addition, the Java test suite for numerical performance tests, as described in [4, 5], will be extended and the acceleration techniques developed by Moreira et al. at the IBM Watson Research Center, [6, 7], will be investigated and eventually implemented in the solver class for integral conservation laws. In addition, the influence of the operating system on the execution speed will be investigated. It has been observed that, for instance, the Solaris operating system, constantly improved the thread handling, and thus led to better multithreading performance.

6.0.2 Thread Scaling and Network Communications Performance

The main emphasis of this activity is to investigate the parallel efficiency of the Java thread concept. For up to 32 processors, used on the HP-V class machine, excellent parallel efficiency was obtained for a sufficient number of threads and sufficient computational work within a *thread*. With the scientific and engineering problems that we have in mind, in particular fluid dynamics or bioscience, these requirements are always met. For instance, a calculation of the flow past a 500 block X-33 configuration or a 5,500 block Ariane 5 launcher utilizes several million grid points. Here, the use of hundreds or even thousands of processors would be justified. Thus the scaling for a variety of architectures will be investigated using up to 128 processors. In addition, the I/O performance across the network for such a large application will be investigated.

6.0.2.1 Mandelbrot Set

This code tests the self-scheduling of threads but also provides uneven loads that is, the computing time for each subdomain depends upon its position and thus requires dynamic load balancing in order to achieve 100% CPU load. The thread scheduling is left to the Java VM and the operating system.

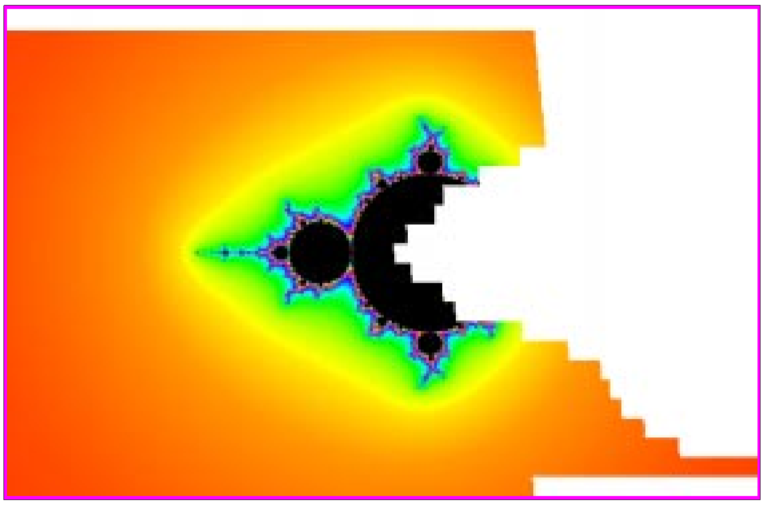

Figure 4 Computation of the Mandelbrot set. The solution domain is performed on a rectnagular grid in the complex plane, subdivided into a number of rows, each handled by a thread.

JDK Release	*VM type*	*warmup*		*production*		
		time in sec	*cpu load*	*time in sec*	*cpu load*	*cpu time*
jdk 1.1.8	jit	2.365	75.00%	2.275	75.00%	1.706
jdk 1.2.2_07	jit	6.576	50.00%	6.321	50.00%	3.161
jdk 1.3.0_02	hotspot - client	68.308	25.00%	7.115	25.00%	1.779
	hotspot - server	2.001	95.20%	1.403	95.20%	1.336
jdk 1.3.1rc2	hotspot - client	82.550	25.00%	2.844	96.10%	2.733
	hotspot - server	2.674	95.30%	1.694	95.30%	1.614

Table 4 Mandelbrot set (600x400 picture size, max iteration 3,000, 200 threads) on a 4 CPU (400MHz) Sun Enterprise 450.

JDK Release	*VM type*	*warmup*		*production*		
		time in sec	*cpu load*	*time in sec*	*cpu load*	*cpu time*
jdk 1.1.8	jit	896.112	75.00%	895.203	75.00%	671.402
jdk 1.2.2_07	jit	880.377	88.60%	873.576	88.60%	773.988
jdk 1.3.0_02	hotspot - client	2708.370	25.10%	2674.853	25.10%	671.388
	hotspot - server	473.793	96.10%	457.965	96.10%	440.104
jdk 1.3.1rc2	hotspot - server	288.140	95.30%	297.660	95.30%	283.670

Table 5 Mandelbrot set (4096x3072 picture size, max iteration 3,000, 1,024 threads) on a 4 CPU (400MHz) Sun Enterprise 450. It seems that only the new jdk 1.3.1lrc2 hotspot client uses native threads.

6.0.2.2 Forward Facing Step

Euler compuatations for the forward facing step at Mach are performed.

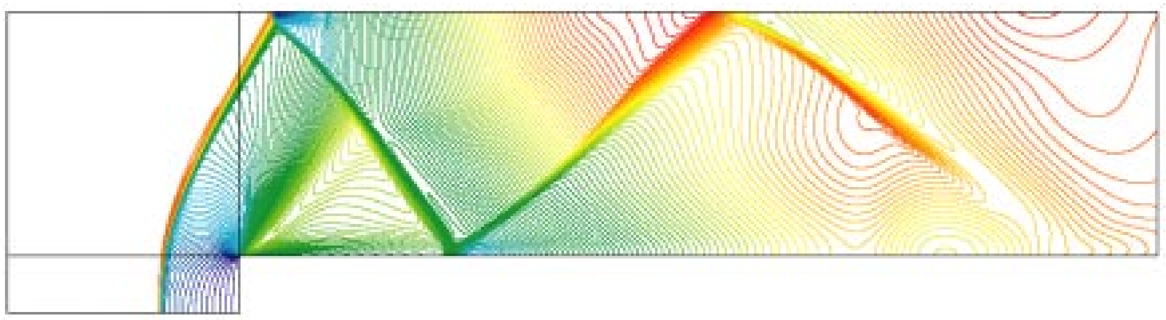

Figure 5 This Testcase is a Mach 3 Euler flow past a forward facing-step. The simulations are based on structured multi-block grids. The computation is explicit and first order accurate, and thus does not too well resolve the expansion region. Shown is the Mach-number distribution

number of blocks	number of cells	time in seconds		parallel speedup
		processor	multi processor (16)	
16	121,104	3,246.73	541.13	6.00
16	200,704	6,908.88	1,077.20	6.41
16	484,416	12,905.88	2,720.48	4.74
48	118,803	2,980.93	225.76	13.20
48	202,800	5,190.54	436.09	11.90
48	480,000	12,663.30	1,162.54	10.89

Table 6 HP V-Class: computing times for 320 iterations

		single processor	multi processor (4)	
16	121,104	852.79	258.26	3.30
16	200,704	1,571.76	532.74	2.95
16	484,416	4,277.96	1,593.45	2.68
48	118,803	756.24	195.95	3.86
48	202,800	1,409.96	378.61	3.72
48	480,000	3,892.67	1,077.06	3.61

Table 7 Sun Enterprise 450: computing times for 320 iterations

6.1 Internet Based Visualization and Navigating Package

Representation of 3D surfaces is a very challenging issue. *JavaGrid* is capable of representing discrete surfaces in discrete triangular or quad format as generated by many gridgenerators, in particular GridPro. The special loader written, provides a way to visualize and investigate complex geometries with a thin client; that is, a machine with just a normal web browser and a reasonably fast connection to the internet. The client is not assumed to have expensive and complex visualization software installed. The files representing the simulation data, as well as the visualization software, are installed on the more powerful server machine.

In *JavaGrid* remote data visualization along with data compression and feature extraction as well as remote computational steering is of prime importance. Since *JavaGrid* allows multiple sessions, multiuser collaboration is needed. Different visualization modules are needed, but here a computational fluid dynamics (CFD) module that allows the perusal of remote CFD data sets was developed, based on the *Java3D* standard.

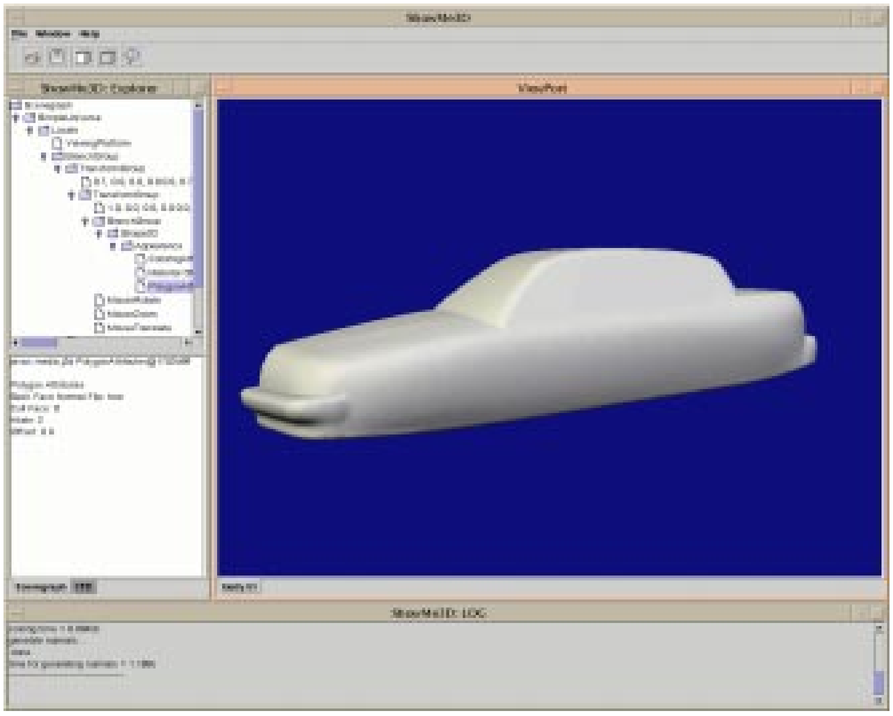

Figure 6 The Virtual Visualization Toolkit (VVT/ShowMe3D) which is part of JavaGrid, depicting a shaded triangulated surface of a generic car.

In large simulations, grids with million of cells are computed, producing hundreds of megabytes of information during each iteration. Depending on the scheme, several thousand iterations may be needed either to converge to a steady state solution or to simulate a time-dependent problem. Hence a fast connection is needed to move data to the client where it can be analyzed, displayed or interacted with to navigate the parallel computation on the server. Therefore a visual interactive package,termed the *Virtual Visualization Toolkit* (VVT) is provided.

A suitably authenticated client sends a request, and this is translated by the server into a response that may consist of several image files linked together by an index page that provides captions and other metadata. The request that is sent to the server is an XML document that instructs the visualization software, that may contain file names, filtering commands, and the type of visualization software that is to be used. We are considering both Tecplot and Ensight for this role. The bulk of the request is in the scripting language used by the chosen software, containing camera angles, isosurface values, colors, and so on; all the information required to build one or more images of the flow.

Clients with more powerful machines and/or a high bandwidth connection to the server might like more than images. We would also like to consider sending back to the client a X3D/VRML file (Extensible 3D the next-generation Virtual Reality Modeling Language based upon XML the Extensible Markup Language). This contains a three-dimensional description of space, rather than just a two-dimensional image. Viewers are available as a plug-in to a web browser (eg. Xj3D or Cosmo player), and the x3d package of *Java3D* now contains a VRML loader. A client could, for example, select a density isosurface value, and have the complete surface returned as a X3D/VRML file, which can then be interactively rotated, zoomed, and viewed within the client's web browser. The intellectual challenge of this work is to provide the client with a way to effectively form the request. This would take the form of a dialogue. Initially, there could be a choice of servers and the CFD files they contain; when a geometry is chosen there might be a choice of flight configurations and flow variables. Once a particular simulation is chosen, then thumbnail views could be displayed, generated either as part of the metadata or generated dynamically. The client can then change parameters with sliders and buttons, and rotate the camera angles through a small X3D/VRML model of the chosen configuration. The client can think of the request that he is generating as a multi-page form that he can adjust by going forward or back. The client can also request the XML document corresponding to the request, for storage or editing.

Once the request is complete, it can be sent to the server for conversion to a visual response by opening the relevant files by the VVT.

REFERENCES

[1] *The Need for Software*, Scientific Computing World, August-September 2000, Issue 54, pp.16.

[2] *Science and technology Shaping the Twenty-First Century*, Executive Office of the President, Office of Science and technology Policy, 1997.

[3] Foster, Ian (ed.): *The Grid: Blueprint for a new Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.

[4] Häuser, J., Ludewig, T., Williams, R.D., Winkelmann R., Gollnick T., Brunett S., Muylaert J.: *A Test Suite for High-Performance Parallel Java*, Advances in Engineering Software, 31 (2000), 687-696, Elsevier.

[5] Ginsberg, M., Häuser, J., Moreira, J.E., Morgan, R., Parsons, J.C., Wielenga, T.J. .: *Future Directions and Challenges for Java Implementations of Numeric-Intensive Industrial Applications*, 31 (2000), 743-751, Elsevier.

[6] Moreira, J.E., S. P. Midkiff, M. Gupta, *From Flop to Megaflop: Java for Technical Computing*, IBM Research Report RC 21166.

[7] Moreira, J.E., S. P. Midkiff, M. Gupta, *A Comparison of Java, C/C++, and Fortran for Numerical Computing*, IBM Research Report RC 21255.

[8] Häuser J., Williams R.D, Spel M., Muylaert J., *ParNSS: An Efficient Parallel Navier-Stokes Solver for Complex Geometries*, AIAA 94-2263, AIAA 25th Fluid Dynamics Conference, Colorado Springs, June 1994.

[9] Häuser, J., Xia, Y., Muylaert, J., Spel, M., *Structured Surface Definition and Grid Generation for Complex Aerospace Configurations*, In: Proceedings of the 13th AIAA Computational Fluid Dynamics Conference -Open Forum, June 29 - July 2, 1997, Part 2,

pp. 836-837, ISBN 1-56347-233-3.

[10] Häuser J., Williams R.D., *Strategies for Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines*, Int. Journal for Numerical Methods in Fluids 15,51-58., John Wiley & Sons, June 1992.

[11] Häuser, J., Ludewig, T., Gollnick, T., Winkelmann, R., Williams, R., D., Muylaert, J., Spel, M., *A Pure Java Parallel Flow Solver,* 37th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 99-0549 Reno, NV, USA, 11-14 January 1999.

[12] Winkelmann, R., Häuser J., Williams R.D, *Strategies for Parallel and Numerical Scalability of CFD Codes, Comp. Meth. Appl. Mech. Engng.*, NH-Elsevier, 174, 433-456, 1999.