American Institute of Aeronautics and Astronautics

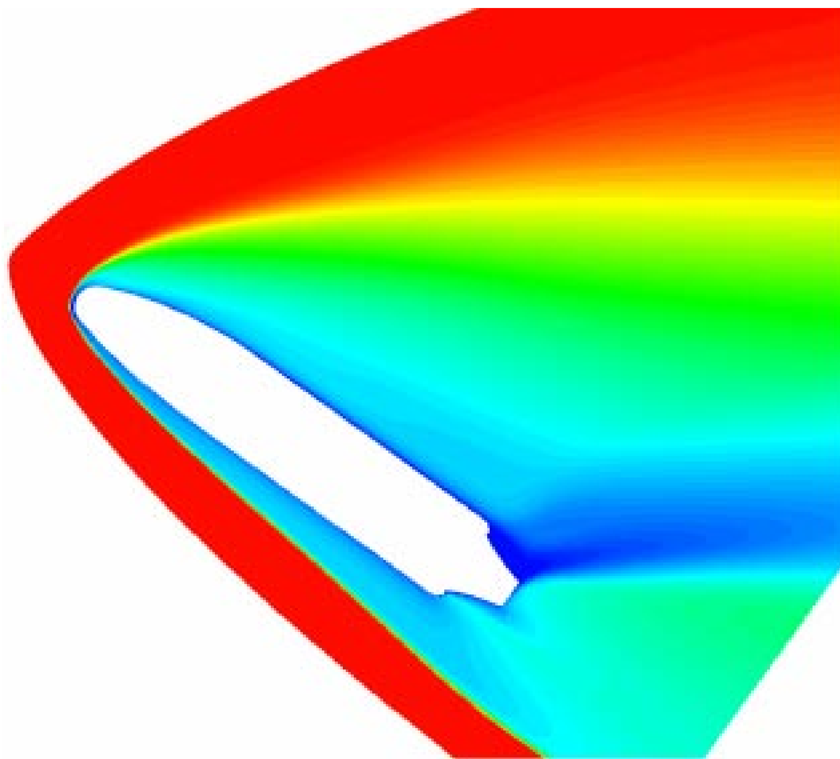
AIAA 99-0549

A Pure Java Parallel Flow Solver

Jochem Häuser, Thorsten Ludewig
Torsten Gollnick, Ralf Winkelmann
Department of Transportation, University of Applied Sciences
and
Department of High Performance Computing, CLE GmbH, Germany

Roy Williams
CACR, California Institute of Technology, Pasadena, CA

Jean Muylaert, Martin Spel
European Space Agency, ESTEC, Noordwijk, NL



X-33 Computation performed by the ParNSS code

37th Aerospace Sciences Meeting and Exhibit

January 11-14, 1999, Reno, NV

Abstract[1]

In this paper an overview is given of the "Have Java"[2] project to attain a pure Java parallel Navier-Stokes flow solver (*JParNSS*) based on the thread concept and remote method invocation (RMI). The goal of this project is to produce an industrial flow solver running on an arbitrary sequential or parallel architecture, utilizing the Internet, capable of handling the most complex 3D geometries as well as flow physics, and also linking to codes in other areas such as aeroelasticity etc.

Since Java is completely object oriented the code has been written in an object-oriented programming (OOP) style. The code also includes a graphics user interface (GUI) as well as an interactive steering package for the parallel architecture. The Java OOP approach provides profoundly improved software productivity, robustness, and security as well as reusability and maintainability. OOP allows code construction similar to the aerodynamic design process because objects can be software coded and integrated, reflecting actual design procedures. In addition, Java is the programming language of the Internet and thus Java objects on disparate machines or even separate networks can be connected.

We explain the motivation for the design of *JParNSS* along with its capabilities that set it apart from other solvers. In the first two sections we present a discussion of the Java language as the programming tool for aerospace applications. In section three the objectives of the Have Java project are presented. In the next section the layer structures of *JParNSS* are discussed with emphasis on the parallelization and client-server (RMI) layers. *JParNSS*, like its predecessor *ParNSS* (ANSI-C), is based on the multiblock idea, and allows for arbitrarily complex topologies. Grids are accepted in *GridPro* or *Plot3D* format. Using *GridPro* property settings, grids of any size or block number can be directly read by *JParNSS* without any further modifications, requiring no additional preparation time for the solver input. In the last section, computational results are presented, with emphasis on multiprocessor Pentium and Sun parallel systems run by the Solaris operating system (OS).

1. The title page shows the Ma number distribution computed by the C-code ParNSS. The JParNSS code is currently restricted to 2D.
2. The HavaJava project aims at a pure Java parallel 3D flow solver and is supported by the Ministry of Science and Culture, Hannover, Lower Saxony, Germany.

1. Java as a Programming Tool in Engineering

Java, introduced in late 1995, was an instant success with the Internet programming community. However, Java was not considered to be a suitable language for software engineering in aerospace, and in particular for computational fluid dynamics (CFD). The reason simply was that Java was lacking performance, because it was an interpreted language.

Recently, however, work at IBM and elsewhere [1], [2], [3] has shown that Java can provide high performance, through careful relaxation of Java's careful exception handling and transformations of the bytecode. .

In early 1997, JavaSoft [4] released a major upgrade to Java, called Java 1.1, that proved to be far superior in overall performance and speed than its predecessor, because of the introduction of advanced concepts (e.g. event handling) as well as the JIT (Just In Time) compiler. In the meantime, Java 1.2 has been released, and Java 2.0 is expected for early 1999. Java 1.1 is a powerful OOP programming language that addresses every kind of programming the (aerospace) engineer might need. In addition, in 1997 the Java Workshop version 2.0 was released by Sun Microsystems, one of an increasing number of Java development environments providing a cross platform development environment along with debugging tools, editor and project management. The major advantage is, however, that an interactive graphical GUI (graphics user interface) is included that greatly alleviates the tedious task of explicitly coding such an interface. The user graphical input is directly converted into Java code that can be combined with the engineering Java code. Furthermore, Java WorkShop also allows creation of Internet applications and web pages.

We believe that Java will profoundly change software development in the engineering area, provided that engineers are able to tap the vast resources of Java and to harness advanced concepts like classes, threads, serialization, or remote method invocation. The reasons why Java should be considered a leapfrog technology for aerospace software development are discussed in the following section.

1.1 Java for Aerospace Applications

CFD along with wind tunnel and flight testing is a respected analysis tool in many areas of aerospace design. CFD is of particular importance for high speed flows where wind tunnel data are difficult to obtain, or when the impact of a single physical phenomenon has to be investigated. Aerospace design of today is marked by diminishing margins that can make or break a design. For instance, the X-33 vehicle, a technology demonstrator for

the next generation space transportation system, does not have to reach LEO (Low Earth Orbit). Therefore, weight issues for the X-33 are not as critical as for the full scale SSTO (Single Stage To Orbit) vehicle, the Venture Star. Consequently, CFD calculations for X-33 can be minimized, concentrating on TPS (Thermal Protection System) design and the aerospike propulsion unit. However, even with the successful demonstration of X-33, there is no guarantee that Venture Star will be a successful spacecraft. Design margins have to be cut to the bare minimum in order to reach LEO with an advertised payload of some 25 metric tons.

Here, in our opinion, CFD will have a crucial role to optimize a given configuration with regard to aerodynamic stability and heat load. Many computer runs have to be performed for the Venture Star configuration, necessitating the handling of very complex grids, requiring short turn around times from CAD to visualized flow solution. In order to reduce computing time, parallel computer architectures have to be used. In the past, much emphasis has been given to the issue of computing time. However, the focus should be on turn around time, that is, taking human interaction out of the loop, starting with the CAD data, generating the grid, producing a CFD solution, visualizing the flow solution, and finally attaining the modified CAD data.

1.2 "Have Java" Objectives

So far, software for computational aerodynamics has been written mainly in Fortran, and in recent years the more advanced C programming language has been employed for visualization tasks. Unfortunately these procedural languages force the programmer to think like a computer, breaking the problem down into a set of basic data types. Object-oriented languages, on the other hand, allow programmers not only to think more efficiently, but also to collaborate more effectively with others. Aerospace engineers are dealing with components like wing, fuselage, nacelle, pylon, engine etc. These components and their properties are difficult to represent in a language like Fortran that only knows integers and reals. This is definitely not the way designers think as they conceive new aircraft or spacecraft.

With the increasing size of aerodynamic codes, robustness and security of the software has become an issue. The recent loss of Ariane 5, flight 501, was attributed to a programming snag, based on a lack of code security, or, in OOP terminology, the lack of *encapsulation*, i.e. preventing other parts of the code to arbitrarily setting values of variables that should be outside their range.

Some of the problems that are inherent in Fortran and to a somewhat lesser degree in C, are: high software cost, low software productivity, insecure code, reduced maintainability, low reusability, poor portability, lack of architecture independence, and clumsy and unfriendly user interfaces. Moreover, to produce these large, integrated applications in a reasonable time and cost requires collaborative engineering effort, involving many companies. We now discuss the features of a computer language that should alleviate some of these problems.

A key component in reaching these objectives is a programming language reflecting the design process, i.e. that allows for the creation of objects that are stand-alone, dedicated to a single task and can be replaced without affecting the rest of the code. The Java programming language can be used for this kind of problem, on any kind of sequential or parallel architecture, and providing independence of the underlying operating system (OS): It can express the mathematical formulas of Fortran, improve on the functionality of C, provide the high-level "object oriented" abstractions of Smalltalk, and Java avoids the obscurity of the C++ language. The error prone concept of pointers to pointers to pointers is not present, and Java has an effective dynamic memory management.

1.3 Java Technologies

For our objectives, we need certain software technologies, some of which may not be well-known in the aerospace community. Java seems to be the only programming framework that provides all of them: the list below summarizes some of the terminology:

1.3.1 Object-Oriented Programming

One of the most important factors is the construction of classes and objects. A class is a template, or blueprint for an object: thus there may be many objects of a given class. A class is the combination of data structures, methods (functions in Fortran and C) that perform operations on the data structures and fields (variables in Fortran and C), and the fields (variables) of this class. Objects provide *inheritance*: given an object 'Engine', for example, with certain properties and methods, we can define a new class 'JetEngine' that inherits from Engine (after all, a Jet engine is a type of Engine). All the properties and methods for Engine work just as well for JetEngine, though some may be implemented differently. Another valuable property of objects is *information-hiding*: the complexity of an object may be only exposed through a simple interface, so that the object is easy to use and understand. A wristwatch is like this -- it has a complex internal structure, but the display of the time is a simple interface.

1.3.2 Robustness

Inevitably, things sometimes go wrong during the flow simulation: files missing, bad grid cells, arithmetic errors, unphysical values, dropped network connections, etc. etc.

Java has a rigorous way to classify and handle such exceptions, coercing the programmer to think about these things while writing the code.

1.3.3 Concurrent, distributed, parallel

Connecting Java objects across disparate machines and networks or running Java code on sequential or parallel architectures is essential to provide the raw computing power needed in the analysis as well as in the design cycles for new air- or spacecraft. Location and type of computer hardware as well as operating system issues should be totally irrelevant to the user, and he should not be even aware of the kind of architecture being used as long as the necessary computing power is provided.

1.3.4 Portability

Most languages are compiled directly to the machine code of the machine on which they are to run, meaning that there can be many versions of the executable, one for each machine. The addition of software and compiler versions to this can make distribution quite difficult. The Java compiler, on the other hand, generates a neutral file format (extension .class), so called *byte code*, from the Java code (extension .java) that is executable on any machine that provides the Java Virtual Machine software, which is practically universally available, translating the bytecode in native machine code.

1.3.5 Leveraging Business Investment

Programs written in Java can take advantage of the huge investment in the language by the commercial world. In particular, there are high-quality, free security packages available to provide authentication and encryption services across a distributed network. We can use commercial Java-based collaboration tools to allow geographically-distributed groups of engineers to work together. We can use web technology to allow engineers to run simulations on the supercomputer without the arcane knowledge of the system that is currently necessary.

1.3.6 Multithreading

In Java, concurrency is achieved via the thread concept. The thread concept is best explained by a simple example: consider a TV-screen that posts several channels at the same time, each shown in a separate small rectangular window. Although these windows (threads) are independent, they are part of the main screen (process), i.e. they share the same address space. Threads are run concurrently, the mapping of threads to processors as well as the scheduling being done by Java and the OS. Thus we have a way to get dynamic load-balancing of a parallel application without explicitly assigning tasks to processors: a threaded application is said to be *self-scheduling*. Java also provides a mechanism for synchronizing threads and for sending messages between threads. Furthermore we no longer need message-passing libraries such as MPI and PVM to communicate between threads, but we can use shared memory or RMI (Remote Method Invocation) instead.

1.3.7 Dynamic linking

Dynamic linking is the ability for a program to link to external code at runtime. For example, suppose we have a set of linear equation solvers: Gaussian elimination, GMRES, Multigrid, LU-decomposition, etc. Traditionally, all of these are linked into one executable file; whereas dynamic linking allows a new solver to be linked at runtime. Besides reducing code size, this feature allows software components to be replaced and maintained without relinking the entire code.

1.3.8 Remote Method Invocation

With a distributed computing system, for example an engineer at a workstation running a supercomputer simulation, the engineer would like to see the computation just as if it were happening on the workstation. Java RMI is one way to do this: the engineer (client) manipulates objects with a user interface, but the actions he performs (the *method invocations*) are actually performed on objects on the supercomputer (the server). This transparent distribution of the computation and steering are vital if we are to provide both the immediacy of a workstation code with the computational power of the supercomputer.

2. Implementation of the Java Parallel Navier-Stokes Solver

JParNSS is based on the idea of collaborating objects that can be located across the Net. Through the GUI layer, located on the client, the engineer fills in the input data like freestream values, the names of the geometry data files, the grid topology etc. The client object sends a message via a standard protocol to the appropriate object on the server that starts the flow computation. The computed solution, or part of it, is sent back to the server to be processed locally. During the computation the client is informed about the computational progress. The GUI contains a monitor and property editor that allows to suspend the computation on the server, i.e. to stop all processors on the server, send new parameter values to the server that forwards the information to all participating processors and resumes the computation. As will be described in the following sections, this is not a trivial process, but Java provides the model that enables

interobject communication, and allows the generation of distributed objects on a multi-processor machine.

Parallelization is based on the concept of multithreading, that is, within *JParNSS* it appears that multiple tasks are performed at the same time. Each task corresponds to a thread in Java. The solution domain comprises a set of blocks and each block is iterated within a thread. When an iteration has been performed, threads (blocks) exchange information with neighboring threads to update their boundaries. This involves some kind of a synchronization operation for the threads. The parallelization strategy is discussed in [7].

The code comprises four layers: the GUI layer, as explained above; the Remote Method Invocation (RMI) that connects the client-side GUI to the high-performance server in a transparent way; a layer for scheduling and synchronization of the parallel threads; and then the solver objects, each of which is responsible for numerical calculation on a single block of the multiblock grid.

In this paper we will concentrate on the RMI and parallel layers, because these concepts are somewhat alien to the non Java programmer. We have not completely implemented the multiblock flow solver; instead the objective has been to investigate Java and threading as a highly flexible, but also efficient, way to express a flow solver.

2.1 Remote Objects: Client and Server

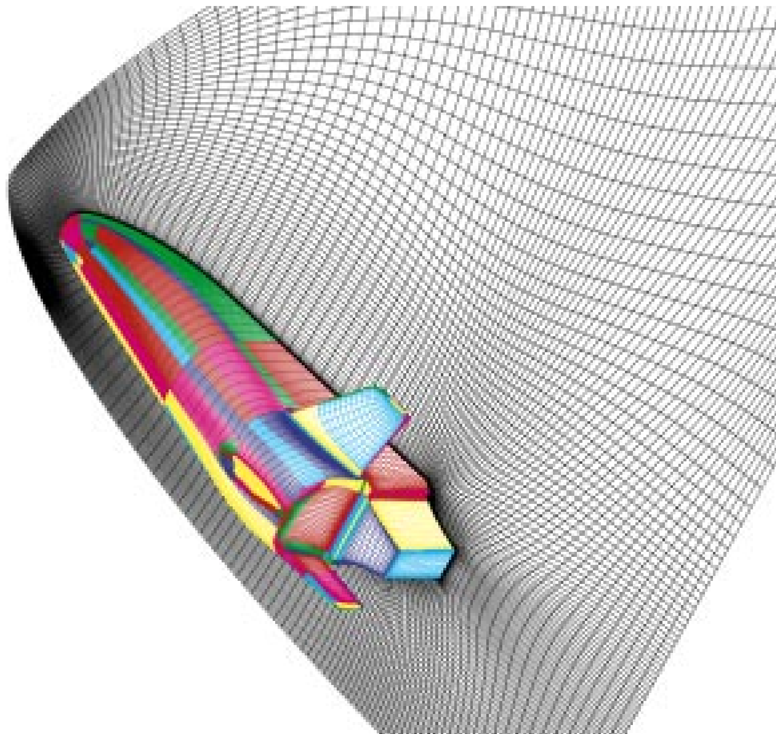


Figure 1: A multiblock grid for the X-33 vehicle. Each block is run in its own thread. Grids may have thousands of blocks, and thus the OS has to create the corresponding number of threads and is also responsible for starting and stopping all threads.

The input data needed to run *JParNSS* is collected on the client and sent across the Net to a server that processes the computational request and starts the parallel code. The server is also responsible for sending back the specified information to the client. The first question to be discussed is how to establish the connection between client and server.

A logical distinction has to be made between the code that resides on the client and code that is on the server. In addition, the common classes that reside on both client and server have to be identified. Furthermore, a decision is needed which of the methods should be local and which have to be remote.

Since we want to send more than just a raw byte stream, a socket connection is not useful. We are using Java's implementation of remote objects that provides the protocol and takes care of all the encoding and decoding: we can invoke remote methods on the server from the client, asking the remote method to return an object to the client. For remote method invocation (RMI) the client machine calls a remote method of an object located on the server.

The *JParNSS* code comprises the modules *Client* and *Server*, as illustrated in Figure 1. The programming on the

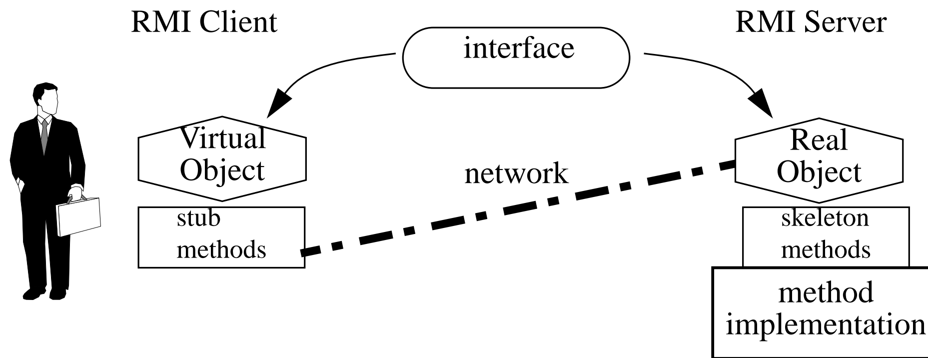


Figure 2. Client-Server communication through Java RMI (Remote Method Invocation). Each shared object has an interface, common to client and server sides, that defines what methods are available. The client can invoke methods on the object, but these are executed on the server where the object actually resides.

client side can manipulate and display objects as if they were local, allowing interactive steering of the code as it runs, and displays showing the progress of the simulation. But in fact, the object is stored on another machine, the high-performance server that is running the simulation. To make this effective, it is important that client and server agree on what methods may be invoked on the object; this occurs because both see the same *interface* file. An interface, in Java, is simply a listing of what methods, with what arguments and return values, can be invoked on the objects of that class; i.e. the name of the class along with the *signature* (name and parameters) of the methods is specified, but none of the methods is actually implemented, which is left to the server code. Each interface for a remote object extends the *Remote* interface of the *java.rmi* package. Each method in such an interface throws a *RemoteException,* that is an error condition is generated in case the remote call fails, for instance, because the network is down etc. This exception (error) has to be caught by all methods and appropriate action has to be taken.

When the client process invokes a method on some object, the Java system encodes the method name and its arguments into a byte-stream, and transfers the request to the server, where it is decoded and actually executed. The communication between client and server is realized by the concept of stub and skeleton. Because the communication process is complex, the meaning of stub and skeleton objects is explicitly outlined below.

Stub object: The stub object resides on the client. When the client invokes a remote method on a remote object on the server, the stub provides the device independent encoding for the parameters to be sent, a process called *parameter marshalling*. The stub also unmarshals the return value received form the skeleton code.

Skeleton object: The skeleton object resides on the server. It decodes the parameters, called *parameter unmarshalling*, and sends back the return value of the remote method to the stub, in marshalled form.

In order to run *JParNSS* its client and server modules have to be set up. It should be noted that client and server may be on the same machine. In other words, the proper Java codes must be running on both client and server. In order for client and server to communicate, the TCP/IP protocol must be enabled, even on a single machine.

Finally, we come to the question of how an RMI session is initiated. In order to allow remote method invocation, the server must be running a demon process called `rmiregistry`, that is listening for requests from clients. In addition, a process must be running on the server that registers certain objects with the registry by means of a text string. In the case of JParNSS, a single object of class JpMaster is registered (see Figure 2).

When a client request arrives, it contains a string that identifies the object that the client is requesting; if this string is in the registry, then the corresponding object is returned to the client. As explained above, it is not the complete object that is returned, but only a reference to the object; but the client sees no distinction between remote and local objects.

2.2 Distributed and Shared-Memory Parallelism

In the *distributed memory* model, parallel computation involved dividing the computation between the processors of the parallel machine. Each processor has its own memory, and information is exchanged through messages. If each processor has a predictable, static workload, such

load-balancing is relatively easy, and can be done before runtime. In a more complex, dynamic situation, it becomes much more difficult. Modern flow solvers, such as ParNSS, switch on and off physical and chemical models, solution algorithms, and grid complexity as the shock moves through the domain. In this case, complex dynamic load-balancing algorithms [19] are needed to move computational load from processor to processor.

Now, however, another approach is becoming viable: the parallel processors run a multithreaded program, they share memory, and the machine is *self-scheduling*. Each thread of control is queued for execution, and a manager process decides which thread can run when. To prevent access of two or more threads to the same variable at the same time Java provides an object lock mechanism using the *synchronized* key word.

3. Parallel Structure of the *JParNSS* Code

Fig. 4 shows the structure of the JParNSS code. A client process connects to the RMI registry on the server, and uses the string “JpMaster” to get an object of class **`JpMaster`**. This object is used to generate multiple sessions, so that several clients can be using the system at the same time for independent computations. The code can be used by several users at the same time. The JpMaster has a **`new`** method that creates a new session and returns an integer sessionID. This integer can be used to **`get`** or **`destroy`** an old session.

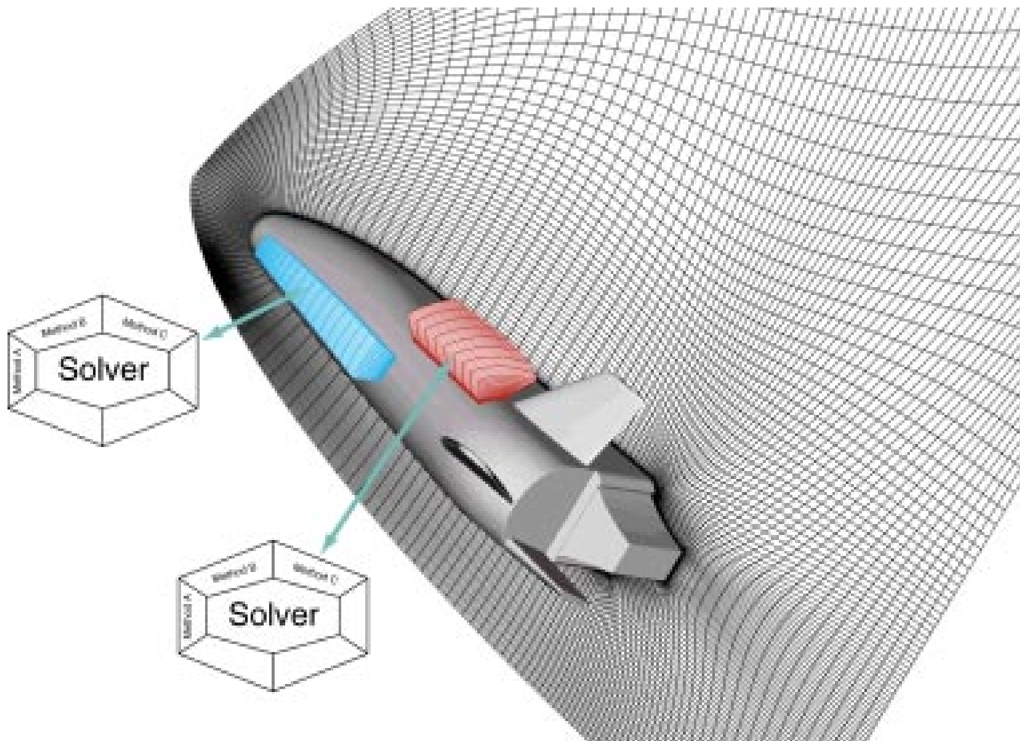


Figure 3. Every solver object contains the data of and the numerics for one block.The solver class is sent from the client to the server that is, different users may use different solvers.

The session object now creates a multithreaded collection of **`Node`** objects, which handle synchronization aspects of the computation; each node is dynamically linked to a **`Solver`** object, which handles the numerics.

The remote Master object is obtained by looking up on the RMI registry on a machine (here called “servername”), asking for an object whose name is “JpMaster”). The Master object can create a new Session by invoking the **`newJpSession`** method; this is of course executed on the remote server because its parent object is remote:

```
mpMaster = (JpMaster)Nam-
ing.lookup(“rmi://servername/JpMaster”);
mpSessionId = mpMaster.newJpSession();
mpSession = mpMaster.getJpSession(
mpSessionId );
```

In this code fragment, note that in addition to the Session object itself, the Master provides an integer **`SessionId`**. This is done so that the client can detach itself from the server and close down, then at a later time, use the **`SessionId`** to reattach to the Session, possibly from a different machine, in order to check on the status of the running computation.

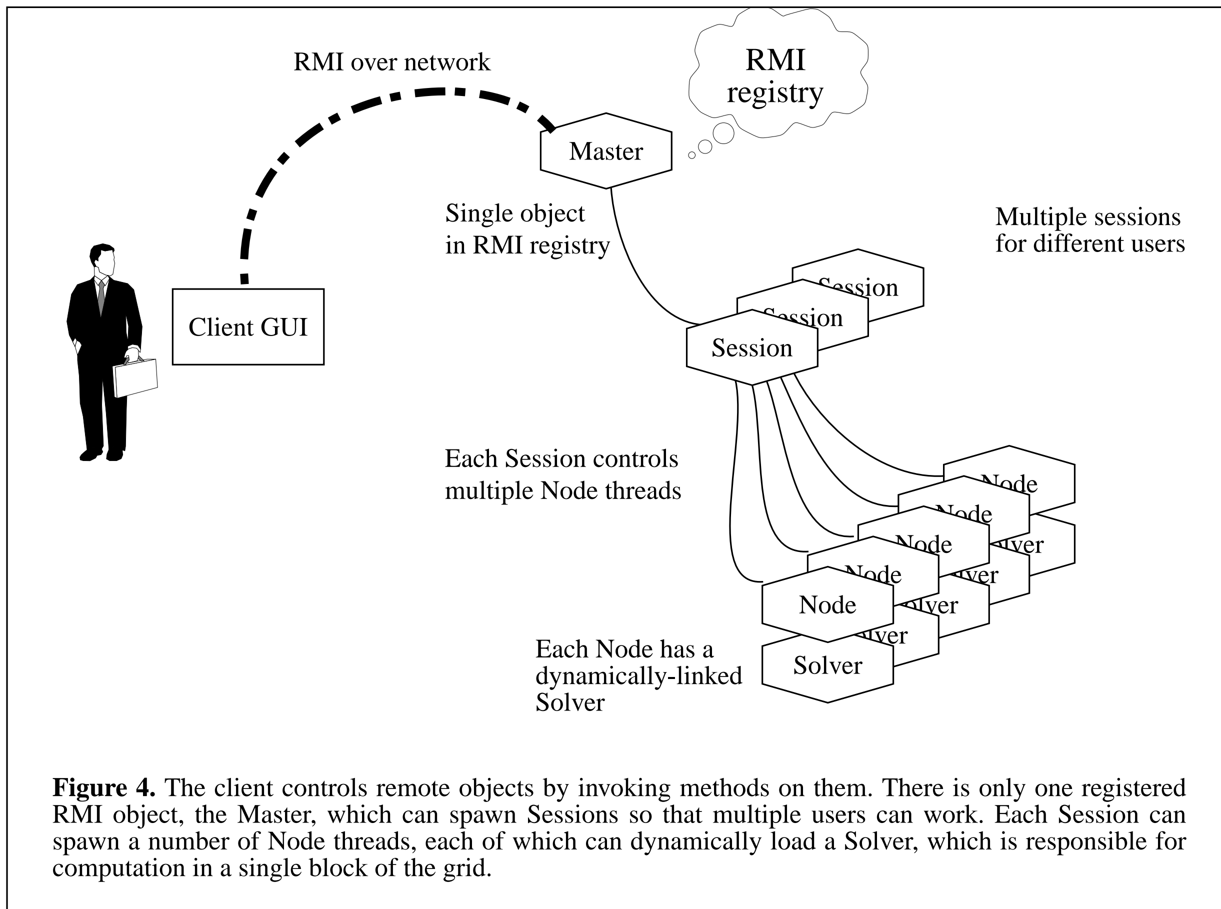


Figure 4. The client controls remote objects by invoking methods on them. There is only one registered RMI object, the Master, which can spawn Sessions so that multiple users can work. Each Session can spawn a number of Node threads, each of which can dynamically load a Solver, which is responsible for computation in a single block of the grid.

Once the Session object is created, it is initialized and its execution begins, meaning that it spawns a number of Node threads, each of which executes a Solver:

```
public class JpSessionImp extends UnicastRemoteObject implements JpSession {
    private JpNodeImp nodeArray[];
    private JpSolver solverArray[];
    public void initSession(int numberOfNodes, ...) {
        nodeArray = new JpNodeImp[numberOfNodes];
        solverArray = new JpSolver[numberOfNodes];
        ...
    }
}
```

The code above is a small part of the process of creating the array of solver threads. Once the arrays of nodes and solvers are created, each is initialized, and each is started. The initialization of a node includes setting up connections to its neighbors, initalizing the data array, and setting up boundary conditions. When all the node threads are ready to run their solvers, computation begins. The solver object then does what we expect in a parallel solver: it exchanges data with the solvers that surround it, and computes the new solution for the next iteration.

3.1 Communication between blocks

Essentially, each block of the computation is alternating between computation and data exchange. 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.

3.2 Running JParNSS

In order to set up the *JParNSS* code on both the client and the server, the following stages are needed.

1. **[Server: compile server programs]** Compile (javac) the java files (extension .java) of the server module on the server.
2. **[Client: compile client programs]** Compile the java files on the client.
3. **[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

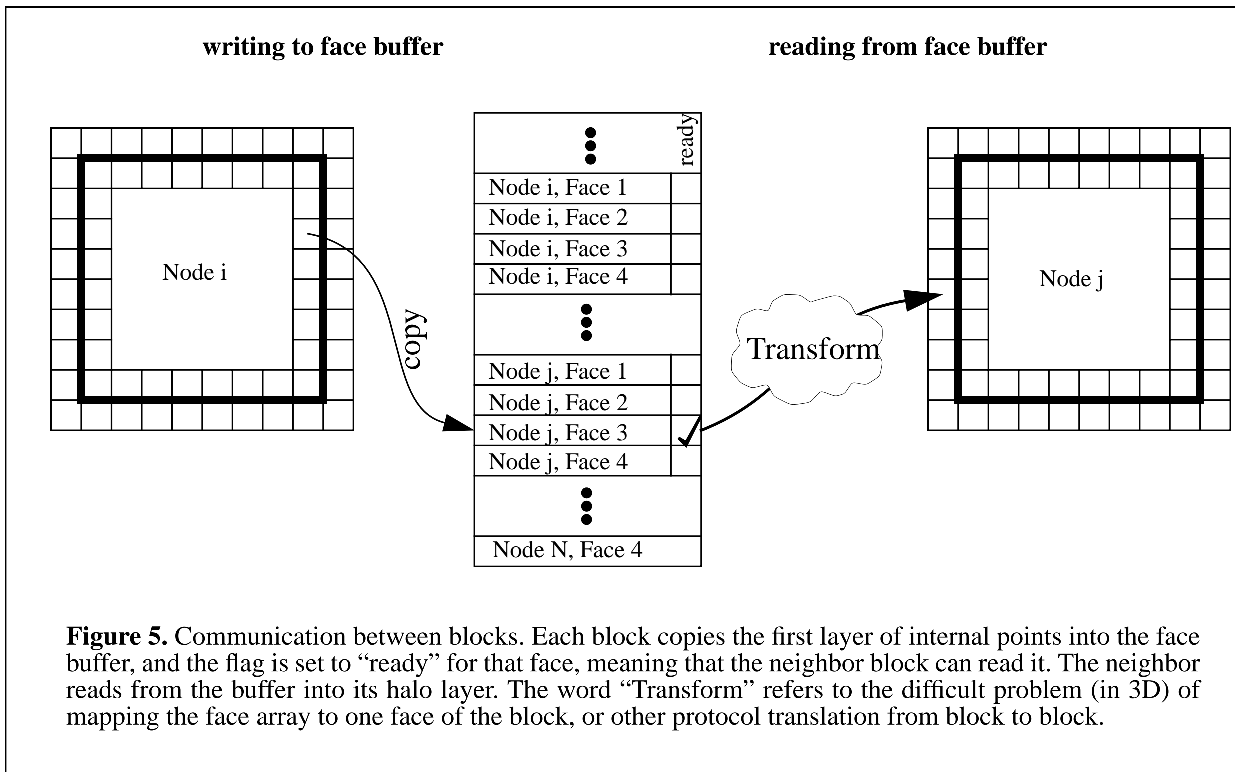


Figure 5. Communication between blocks. Each block copies the first layer of internal points into the face buffer, and the flag is set to "ready" for that face, meaning that the neighbor block can read it. The neighbor reads from the buffer into its halo layer. The word "Transform" refers to the difficult problem (in 3D) of mapping the face array to one face of the block, or other protocol translation from block to block.

stub code contains the signatures of the remote methods and provides the necessary information for the client code.

4. [**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.*
5. [**Server: object registration (binding)**] Start the code that registers (*binding*) all objects of class implementation on the server, i.e. the *JpMaster* process.
6. [**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.

The interfaces, that are seen by both client and server, are stored in directory *Share. Client, Server*, and *Share* are subdirectories of directory *JParNSS* containing the make file, the documentation etc.

4. Implementation of Threads by the Operating System)

There are three basic models of thread implementations: "many-to-one", "one-to-one", and "many-to-many".

In a many-to-one implementation — also referred to as "user-level threads" — all thread activity is restricted to user space. Additionally, at any given time, only one thread can access the kernel, so only one schedulable entity is known to the operating system. As a result, this multithreading model provides limited concurrency and does not exploit multiprocessors.

In the "one-to-one" thread model, the main problem is that it places a restriction on the developer to be careful and frugal with threads, as each additional thread adds more "weight" to the process. Consequently, many implementations of the one-to-one model, such as Windows NT and the OS/2® threads package, limit the number of threads supported on the system. (i.e. 1024 threads on Windows NT)

In the "many-to-many" model, a program can have as many threads as are appropriate without making the process too heavy or burdensome. In this model, a user-level threads library provides sophisticated scheduling of user-level threads above kernel threads. The kernel needs to manage only the threads that are currently active. A many-to-many implementation at the user level reduces programming effort as it lifts restrictions on the number of threads that can be effectively used in an application.

A many-to-many multithreading implementation thus provides a standard interface, a simpler programming model, and provides optimal performance for each process. The Java virtual machine with the Solaris operating environment is the first many-to-many

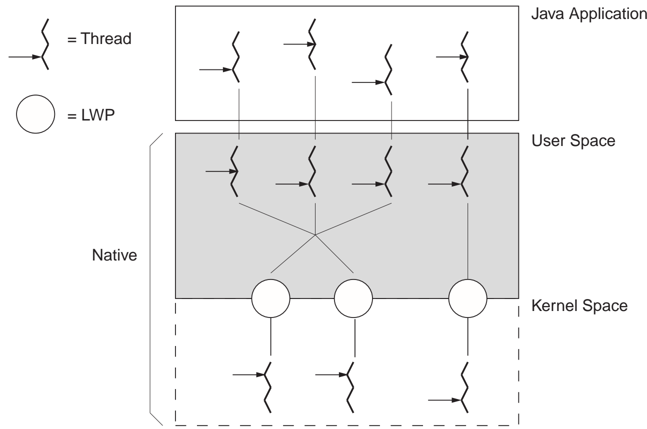


Figure 6. The many-to-many model (many user-level threads to many kernel-level threads) avoids many of the limitations of the one-to-one model, while extending multithreading capabilities even further. The many-to-many model, also referred to as the two-level model, minimizes programming effort while reducing the cost and weight of each thread.

commercial implementation of Java on a multithreaded OS.

5. Computational Results

At present the solver class is implemented for 2D and for Cartesian coordinate systems. In addition, the full multiblock layer has not been implemented. No attempts were made to reduce the computing time, using for example the IBM Java compiler [1], [2] or the software offered by [3]. Neither was there an attempt to replace RMI by a faster communication module such as [18]. The design of JParNSS strictly follows Kernighan's rule '*Make it right before you make it faster*'. In reference [1] speeds between 80% and 90% of corresponding Fortran programs were obtained and a compiler will be available soon.

As a test case, we have chosen to compute an Euler flow past a forward-facing step at Mach 3. The resulting mach-number field is shown in Fig. 7.

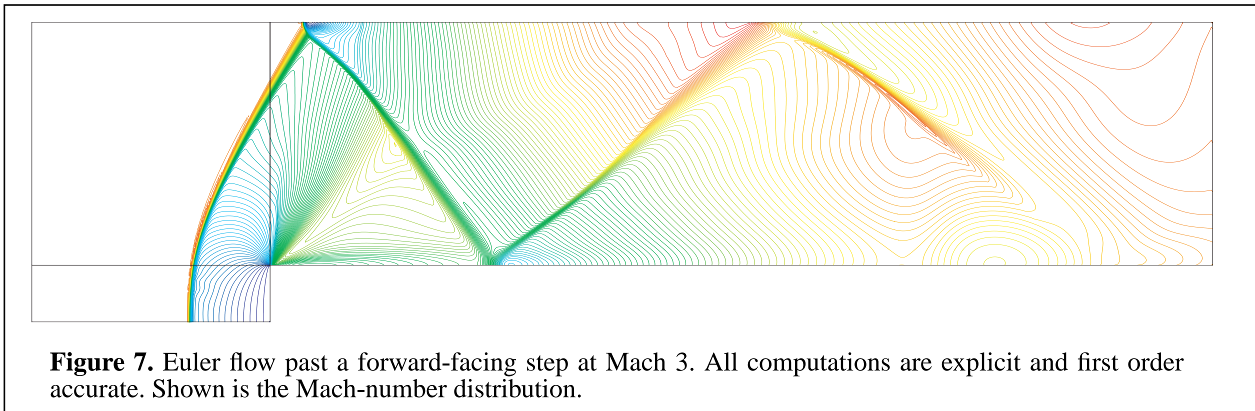


Figure 7. Euler flow past a forward-facing step at Mach 3. All computations are explicit and first order accurate. Shown is the Mach-number distribution.

5.1 Computing Times for Monoprocessors

In Table 1 computing times for various processors are

Table 1. JParNSS computing times for various monoprocessors

Architecture	Number of blocks	Number of cells	Computing time [s]	Memory [MB]	JDK version
PentiPentium II 300	38	12000	331	256	1.1.6

Table 1. JParNSS computing times for various monoprocessors

AMD K6-2 300 MHz	38	12000	1045	64	1.1.7
Sun Ultra 10	38	12000	358	512	1.1.3
Sun E450	38	12000	237	2000	1.1.6
SGI R8000	38	12000	2678	3000	1.1.6
PentiPentium II 300	148	12000	355	256	1.1.6
AMD K6-2 300 MHz	148	12000	1104	64	1.1.7
Sun Ultra 10	148	12000	354	512	1.1.3
Sun E450	148	12000	275	2000	1.1.6
SGI R8000	148	12000	2753	3000	1.1.6

given. As far as possible, the same version of JDK was used.

5.2 Computing Times for Multiprocessors

In Table 2 JParNSS is run on a variety of architectures.

Table 2. JparNSS computing times for multiprocessor architectures

Number of processors	Architecture	Number of blocks	Number of cells	Computing time [s]	Memory [MB]	JDK version
2	Pentium II 300 MHz	3	48000	1273	256	1.1.6
2	Sun Ultra 60	3	48000	787	512	1.2beta5
2	Sun E450	3	48000	1015	2000	1.1.6
3	Sun E450	3	48000	963	2000	1.1.6
4	Sun E450	3	48000	1044	2000	1.1.6
2	Pentium II 300 MHz	10	48000	708	256	1.1.6
2	Sun Ultra 60	10	48000	475	1100	1.2beta5
2	Sun E450	10	48000	563	2000	1.1.6
3	Sun E450	10	48000	472	2000	1.1.6
4	Sun E450	10	48000	455	2000	1.1.6
2	Pentium II 300 MHz	38	48000	658	256	1.1.6
2	Sun Ultra 60	38	48000	417	1100	1.2beta5
2	Sun E450	38	48000	514	2000	1.1.6
3	Sun E450	38	48000	349	2000	1.1.6
4	Sun E450	38	48000	261	2000	1.1.6
2	Pentium II 300 MHz	148	48000	640	256	1.1.6
2	Sun Ultra 60	148	48000	421	1100	1.2beta5
2	Sun E450	148	48000	529	2000	1.1.6
3	Sun E450	148	48000	343	2000	1.1.6
4	Sun E450	148	48000	260	2000	1.1.6
2	Pentium II 300 MHz	38	192000	n.a.	256	1.1.6
2	Sun Ultra 60	38	192000	2926	1100	1.2beta5
2	Sun E450	38	192000	n.a.	2000	1.1.6
3	Sun E450	38	192000	n.a.	2000	1.1.6
4	Sun E450	38	192000	n.a.	2000	1.1.6

Table 2. JparNSS computing times for multiprocessor architectures

Number of processors	Architecture	Number of blocks	Number of cells	Computing time [s]	Memory [MB]	JDK version
2	Pentium II 300 MHz	148	192000	n.a.	256	1.1.6
2	Sun Ultra 60	148	192000	283342	1100	1.2beta5
2	Sun E450	148	192000	n.a.	2000	1.1.6
3	Sun E450	148	192000	n.a.	2000	1.1.6
4	Sun E450	148	192000	n.a.	2000	1.1.6

with different numbers of processors. In all runs the number of threads equals the number of processors. This corresponds to the fact that a grid with several thousand blocks has to be run on multiprocessor system whose number of processors is substantially smaller than the number of blocks. As can be seen from Table 2, the present paper is restricted to architectures with a small number of processors. In Fig. 8 is shown the compute times as the number of blocks of the multiblock grid is varied. Here the number of active threads is the same as the number of blocks. As blocks are split, we have more threads than processors, so that each processor has enough work to do: the computational efficiency increases because the load-balance between the four processors improves. For a much larger number of blocks, however, the thread overhead becomes larger than the computational work associated with the block, and efficiency drops.

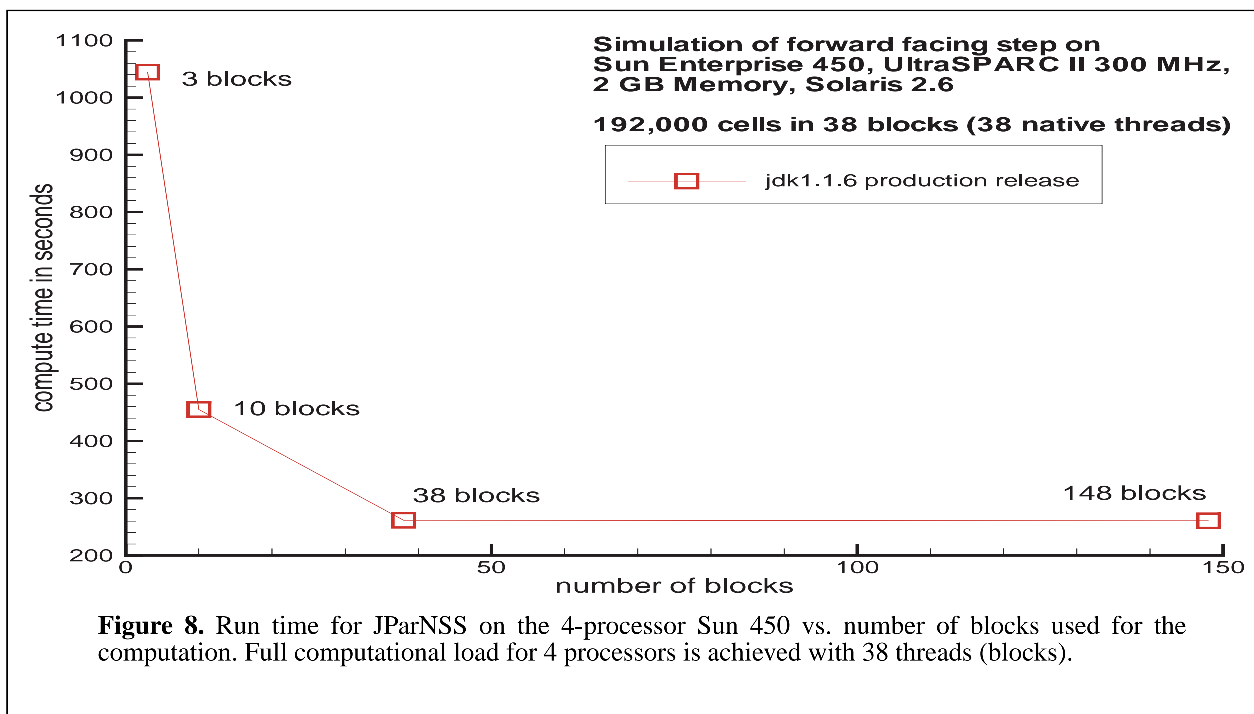


Figure 8. Run time for JParNSS on the 4-processor Sun 450 vs. number of blocks used for the computation. Full computational load for 4 processors is achieved with 38 threads (blocks).

6. Conclusions and Future Work

This prototype implementation of a multiblock solver represents a first stage in the "Have Java" project. We have shown how a flexible, component-based architecture can be used to create a fluid solver. In this paper, we have shown the component that is responsible for parallelism, and also the remote-invocation layer, which allows a remote client to initiate and steer a computation on a supercomputer.

In the component model, different parts of the code can be developed separately. In this context, *component* implies a well-defined interface with other components, so that components can be easily exchanged, upgraded, or worked-on by different parts of a collaboration.

The next component that we shall create in this program is a multiblock component, allowing complex geometries and topologies to be handled. The computational domain is divided logically into a set of boxes, each with its own local coordinate system. The boxes are then connected through the multiblock component.

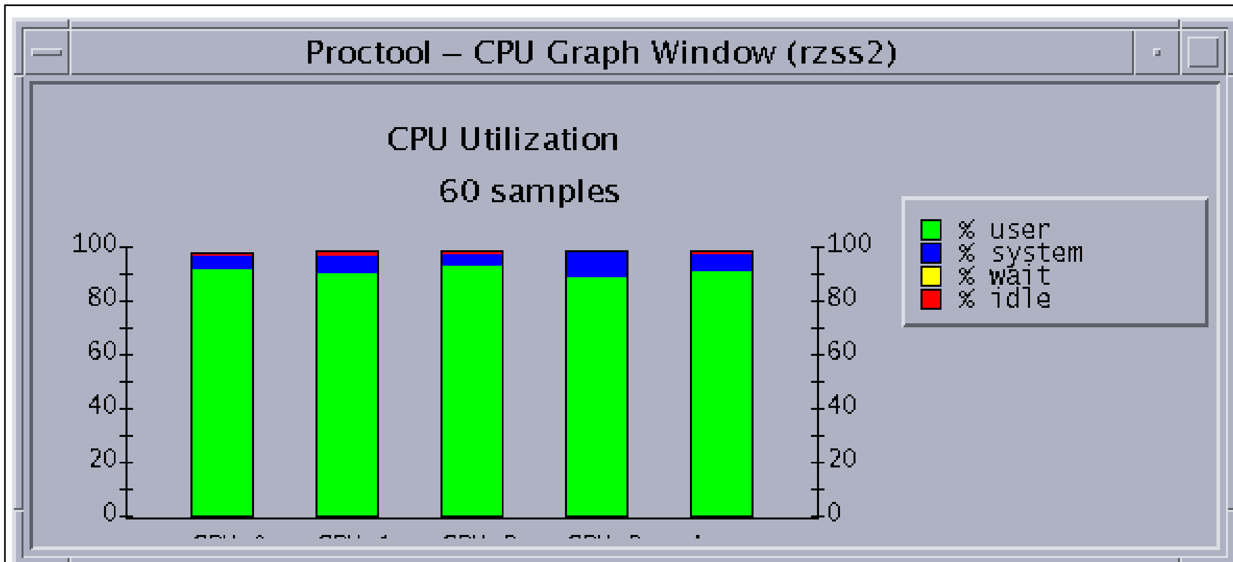


Figure 9. Processor utilization for the 4-processor Sun E450 running JParnss. This screenshot is obtained from the Solaris tool *proctool*. The bars from left to right denote CPU numbers 0 to 3, the rightmost bar shows the average load of the CPUs. It is important that the thread scheduler generates a full computational load for all processors. If the number of threads is too small or threads of highly different compautational load have been produced, parallel efficiency is reduced.

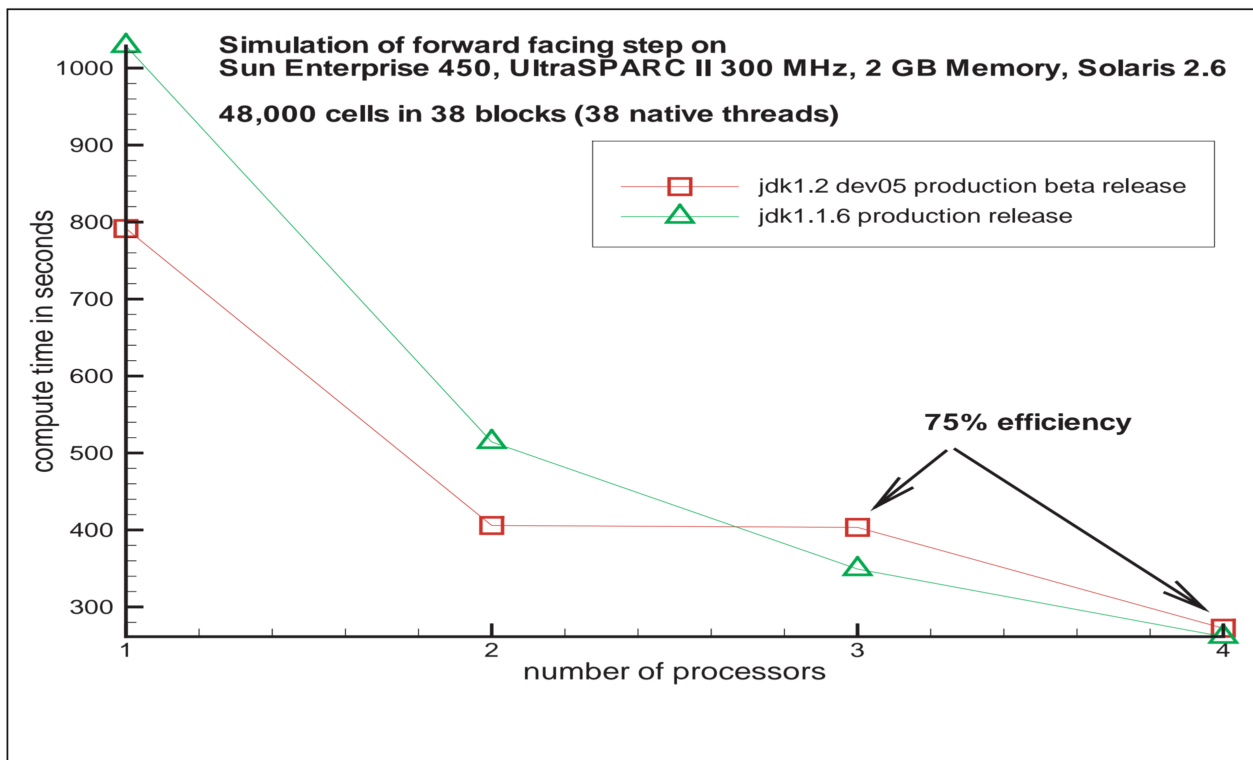


Figure 10. Run time for JParNSS on the 4-processor Sun 450 vs. number of processors. The 38 block grid comprising 48,000 cells was used for the simulations. The figure shows a substantial speedup for the 1 and 2 processor configurations for JDK1.2 over JDK1.1.6. However, when 3 and 4 procesors are used, JDK 1.2 does not scale properly and parallel efficiency is lost.

We have provided a framework for a solver that is cast in the form of general conservation laws. Another component of the system is the expression of the conservation system that is being solved.

Acknowledgments

The authors are grateful to Professor Mark Cross, University of Greenwich, London for numerous stimulating discussions.

This project was partly funded by the ministry of Science and Culture of the State of Lower Saxony, Germany and the European Commission under contract *JavaPar 1998.262.*

References

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

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

[3] Java Lapack implementation, from AppStar LLC., http://www.appstar.com.

[4] JavaSoft Corporation: http://www.javasoft.com/

[5] Horstman, Cay S., Cornell, G., 1998: *CoreJAVA, Volume I-Fundamentals*, Prentice Hall.

[6] Horstman, Cay S., Cornell, G., 1998: *CoreJAVA, Volume II-Advanced Features,* Prentice Hall.

[7] 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

[8] 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

[9] Eiseman, Peter R., 1998: *GridPro v3.1, The CFD Link to Design, Topology Input Language Manual*, Program Development Corporation Inc., 300 Hamilton Ave., Suite 409, White Plains, NY 10601.

[10] Rich, B., 1994: *Skunk Works*, Little Brown and Company.

[11] Fox, G.C. (ed.), 1997: *Java for Computational Science and Engineering- Simulation and Modeling I*, Concurrency Practice and Experience, Vol. 9(11), June 1997,Wiley.

[12] Fox, G.C. (ed.), 1997: *Java for Computational Science and Engineering- Simulation and Modeling II*, Concurrency Practice and Experience, Vol. 9(11), November 1997,Wiley.

[13] Proceedings of the *ACM Workshop on Java for High Performance Network Computing*, Stanford University, Palo Alto, California, February 28 & March 1, 1998, *www.cs.ucsb.edu/conferences/java98.*

[14] Lea, D., 1997: *Concurrent Programming in Java*, Addison Wesley.

[15] Sun Microsystems, *Java Remote Method Invocation Specification*, Revision 1.41, JDK 1.1.1 March 24, 1997, http://www.javasoft.com/docs/

[16] James Gosling, Henry McGilton, *The Java Language Environment - A White Paper*, Sun Microsystems Oktober 1995, http://www.javasoft.com/docs/

[17] SunSoft, Java On Solaris 2.6 - A White Paper, September 1997,
http://www.sun.com/solaris/java/wp-java/

[18] Java Party, an improved remote-method invocation, University of Karlsrühe.

[19] Winkelmann, R., Häuser J., Williams R.D, *Strategies for Parallel and Numerical Scalability of CFD Codes,* Comp. Meth. Appl. Mech. Engng. (accepted)

.