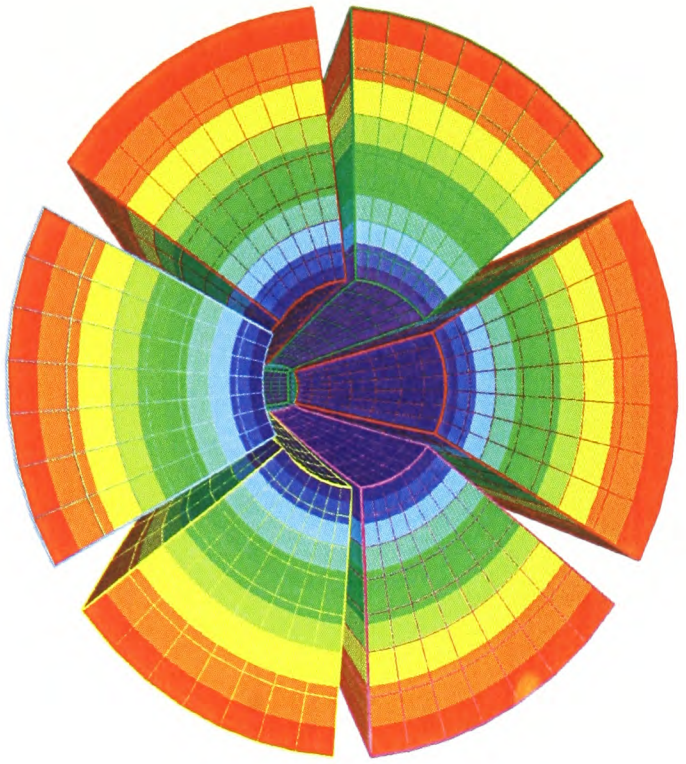
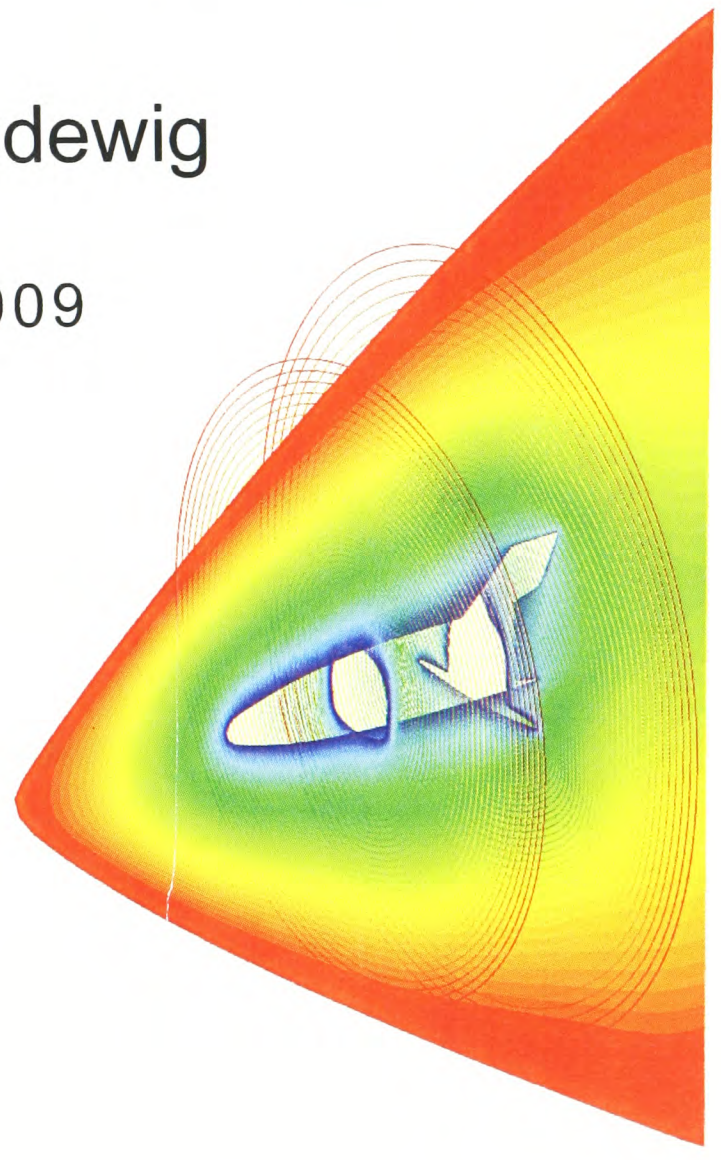


JUSTGrid: A Pure Java HPCC Grid Architecture for Multi-Physics Solvers.

Thorsten Ludewig

August 2009



JUSTGRID

A Pure Java HPCC Grid Architecture
for Multi-Physics Solvers
Using Complex Geometries.

Thorsten Ludewig

UNIVERSITY OF GREENWICH LIBRARY

A thesis submitted in partial fulfilment of the

requirements of the University of Greenwich

for the degree of Doctor of Philosophy

This research programme was carried out in collaboration with the

University of Applied Sciences Braunschweig/Wolfenbüttel

and the

HPCC-Space GmbH, Salzgitter

Germany

August 2009

Abstract

After the Earth Simulator, built by NEC at the Japan Marine Science and Technology Centre (JAMSTEC) on an area of 3,250 m2 (50m×65m), began it's work in March 2002 with the outstanding performance of 35,860 Gflops (40 TFlops peak) [TRI00], numerous scientists opted in favour of such a high-performance computation and communications (HPCC) approach, suggesting to build again Cray type vector supercomputers that dominated scientific computing in the mid seventies. Today (2009) the extended Earth Simulator has a peak performance of 131 TFlops but it was outperformed by several other systems with multi-core[1] architectures. Top 1 in June 2009 is the RoadRunner build by IBM for the DOE/NNSA/LANL with a peak performance of 1456 TFlops. Multi-core processors are now build in every PC for the consumer market and not only for HPC systems. It should be remembered that the computer games industry is responsible for the revolution in high end 3D graphics cards that convert any PC into a most powerful graphics workstation. It should be obvious, despite the computational power of the Earth Simulator, that this definitely is not the road of HPCC for general scientific and engineering computation.

"I hope to concentrate my attention on my research rather then how to program", says
Hitoshi Sakagami, a researcher at Japan's Himeji Institute of Technology and a
Gordon Bell Prize finalist for work using the Earth Simulator [TRI00].

I fully agree with this statement, and this is one of the major reasons that I have chosen Java as high performance computing language. Programming vector computers is a difficult task, and to obtain acceptable results with regard to announced peak performance has been notoriously cumbersome. On the other hand, multi-core systems with many processors on a single chip need to be programmed in a different, namely a multi threaded way. Threads are a substantial part of the Java programming language. Java is the only general programming language that does not need external libraries for parallel programming, because everything needed is built into the language. In addition, there are major additional advantages of the Java language (object oriented, parallelization, readability, maintainability, programmer productivity, platform independence, code safety and reliability, database connectivity, internet capability, multimedia capability, GUI (graphics user interfaces), 3D graphics (Java 3D) and portability etc.) which were discussed in this thesis.

The objective of this work is to build an easy to use software framework for high performance computing dealing with complex 3D geometries. The framework should also take care of all the advantages and behaviours of modern multi-core/multi-threaded hardware architectures. In view of the increasing complexity of modern hardware, working on solutions of multi-physical problems demands for software, that makes the solving process mostly independent of the available machinery.

---

1 A multi-core chip is composed of two or more independent CPUs (cores) in one single processor.

Table of Contents

Illustration Index

Index of Tables

Nomenclature and Constants

A	area
B	magnetic induction field
c	speed of light in vacuum
$c_A = \frac{B}{\sqrt{\mu_0 \rho}}$	Alfven wave speed
$\boldsymbol{D}$	displacement field
$\frac{\partial \boldsymbol{D}}{\partial t}$	displacement current
E	total energy per unit mass or per unit volume (internal plus kinetic plus magnetic energy)
e	internal energy per unit mass
e_e	electric energy density
$e_{em} = e_e + e_m$	electromagnetic energy density
$\hat{\boldsymbol{e}}_i, \hat{\boldsymbol{e}}_j, \hat{\boldsymbol{e}}_k$	unit vectors (Cartesian system in x, y, z direction)
e_m	magnetic energy density
$e_0 = 1.6 \times 10^{-19} C$	electron charge
G_i	electromagnetic momentum density
H	magnetic field
$\boldsymbol{j} = \rho \boldsymbol{v}$	current density
k	thermal diffusivity
$k_B = \frac{R_G}{N_A} = 1.38054 \times 10^{-23} J/K$	Boltzmann constant
$L = \frac{k}{\rho c_p}$	diffusivity
$\boldsymbol{M}$	magnetization per volume

$Ma_m = \frac{v}{c_A}$	magnetic Mach number
M_{ij}	magnetic stress tensor components
$m_e = 9.1 \times 10^{-30} kg$	electron mass
$N_A = 6.023 \times 10^{26} / kmol$	Avogadro number
$N_D = \frac{4\pi}{3} n_e \lambda_D^3 = 1.37 \times 10^6 T \frac{3/2}{n_e^{1/2}}$	number of electrons in a Debye sphere
$\boldsymbol{P}$	polarization per volume
$P_r = \frac{v}{L} = \frac{\mu c_p}{k}$	Prandtl number
q	charge (electric)
$r_{ce} = \frac{v_{e\perp}}{\omega_{ce}} = \frac{m_e v_{e\perp}}{e B}$	electron cyclotron radius (only the velocity component perpendicular to the magnetic field is effective)
$r_{ci} = \frac{v_{i\perp}}{\omega_{ci}} = \frac{m_i v_{i\perp}}{Z e B}$	ion cyclotron radius (only the velocity component perpendicular to the magnetic field is effective)
$Re = \frac{v L}{\nu}$	Reynolds number
$Re_m = \frac{v L}{\eta} = v L \sigma \mu_m$	magnetic Reynolds number
$R_G = 8.31 \times 10^3 J / Kmol \cdot K$	universal gas constant
$\boldsymbol{S}$	Poynting vector
T	temperature
$v = (u, v, w)$ $v = (v_x, v_y, 0) in\ 2D$	velocity vector

Greek Symbols

$\rho = \rho(x, y, z, t)$	mass density
ρ_e	electric charge density

ν	kinematic viscosity $\left[\frac{m^2}{s}\right]$
$\lambda_D = \left(\frac{\epsilon_0 k T}{n_e e^2} \right)^{1/2}$	Debye shielding length
$\mu = \nu \rho$	dynamic viscosity
$\eta = \frac{1}{\sigma \mu_m}$	magnetic viscosity (diffusivity)
$\epsilon_0 \mu_0 c^2 = 1$	
$\frac{1}{\sigma}$	specific resistance
$\Phi_e(A) = \epsilon_0 \int_A \boldsymbol{E} \cdot d\boldsymbol{A}$	electric flux, the area $\boldsymbol{A}$ is bounded by curve C
$\Phi_m(A) = \int \boldsymbol{B} \cdot d\boldsymbol{A}$	magnetic flux (is 0 if A is closed)
$\Phi_e(A) = \int_A \boldsymbol{D} \cdot d\boldsymbol{A}$	if medium is polarized
ϵ_0	permittivity of free space = $8.85 \times 10^{-12} \frac{As}{V m}$
μ_0	permeability of free space = $4\pi \times 10^{-7} \frac{V s}{A m}$
μ_m	magnetic permeability
σ	conductivity $\left[\frac{1}{\Omega_m}\right]$
$\sigma_0 = \frac{n_e e^2}{m_e \nu_e}$	plasma conductivity for direct current
$\omega_{pe} = \left(\frac{n_e e^2}{m_e Z^2 \epsilon_0} \right)^{1/2}$	electron plasma frequency
$\omega_{pi} = \left(\frac{n_i e^2}{m_i Z^2 \epsilon_0} \right)^{1/2}$	ion plasma frequency

$$\omega_{pe}=\left(\frac{n_e e^2}{m_e Z^2 \epsilon_0}\right)^{1/2}$$

$$p_m=\frac{B_0^2}{2\mu_0}$$ magnetic pressure

Characteristic Numbers in Magnetohydrodynamics

Avogadro number

$$N_A=6.023\times 10^{26}/kmol$$

Alfven velocity (wave speed)

$$c_A=\frac{B}{\sqrt{\mu_0 \rho}}$$

Hartmann number

(magnetic body force / viscous force)

$$K_H=(\sigma B^2 L^2/\mu)^{\frac{1}{2}}=(\frac{Re\, Re_m}{Ma_m})^{\frac{1}{2}}$$

Mach number

$$M_a=\frac{v}{a}$$

Magnetic Reynolds number

$$Re_m=\frac{v L}{\eta_m}=v L \sigma \mu_m$$

Prandtl number

$$P_r = \frac{v}{L} = \frac{\mu c_p}{k}$$

Reynolds number

$$Re = \frac{v L}{\eta}$$

Abbreviations

BC	Boundary condition
CFD	Computational Fluid Dynamics
EFA	Electromagnetic Field Actuators
HLL	Harten-Lax-van Leer
HLLC	Harten-Lax-van Leer-Contact discontinuity
IC	Initial condition
MHD	MagnetoHydroDynamics
MPI	Message Passing Interface
MPP	Massively Parallel Processing
JUST	Java Ultra Simulator Technology
PDE	Partial Differential Equation
PVM	Parallel Virtual Machines
SMP	Symmetric MultiProcessing

Acknowledgment

I am most grateful for the opportunity to do this research to *Prof. Dr. Jochem Häuser* and *Prof. Dr. Mayur Patel.* In particular, I gratefully acknowledge the large number of discussions and the substantial support of Prof. Dr. Jochem Häuser.

Many thanks are conveyed to my friends *Torsten Gollnick*, *Olav Rybatzki and Dr. Ralf Winkelmann* for their selfless support.

I would like to thank *Jean Muylaert* ESTEC, ESA, Noordwijk, The Netherlands for his continuous interest and the provision of wind tunnel data.

I am grateful to *Prof. Peter Eiseman*, Program Development Comp., White Plains, New York, USA for numerous stimulating discussions especially on the field of grid generation.

I would like to thank *my parents* for their encouragement and support.

My heartiest thanks go to my godfather *Helmut Scholz* for his unselfish help in a hopeless situation for me.

Last but not least, I would like to thank *my family* for their support and time to complete this thesis.

This work was partly funded by the ministry of Science and Culture of the State of Lower Saxony, Germany under AGIP 1999.365 EXTV program.

JUSTGRID was also part of the following EFRE projects partly funded by the ministry of Science and Culture of the State of Lower Saxony, Germany and the European Commision under contracts JavaPar 1998.262 and *JUST 2002.108*.

1 Motivation of the Thesis

1.1 Challenges and current status of computational simulation for CFD

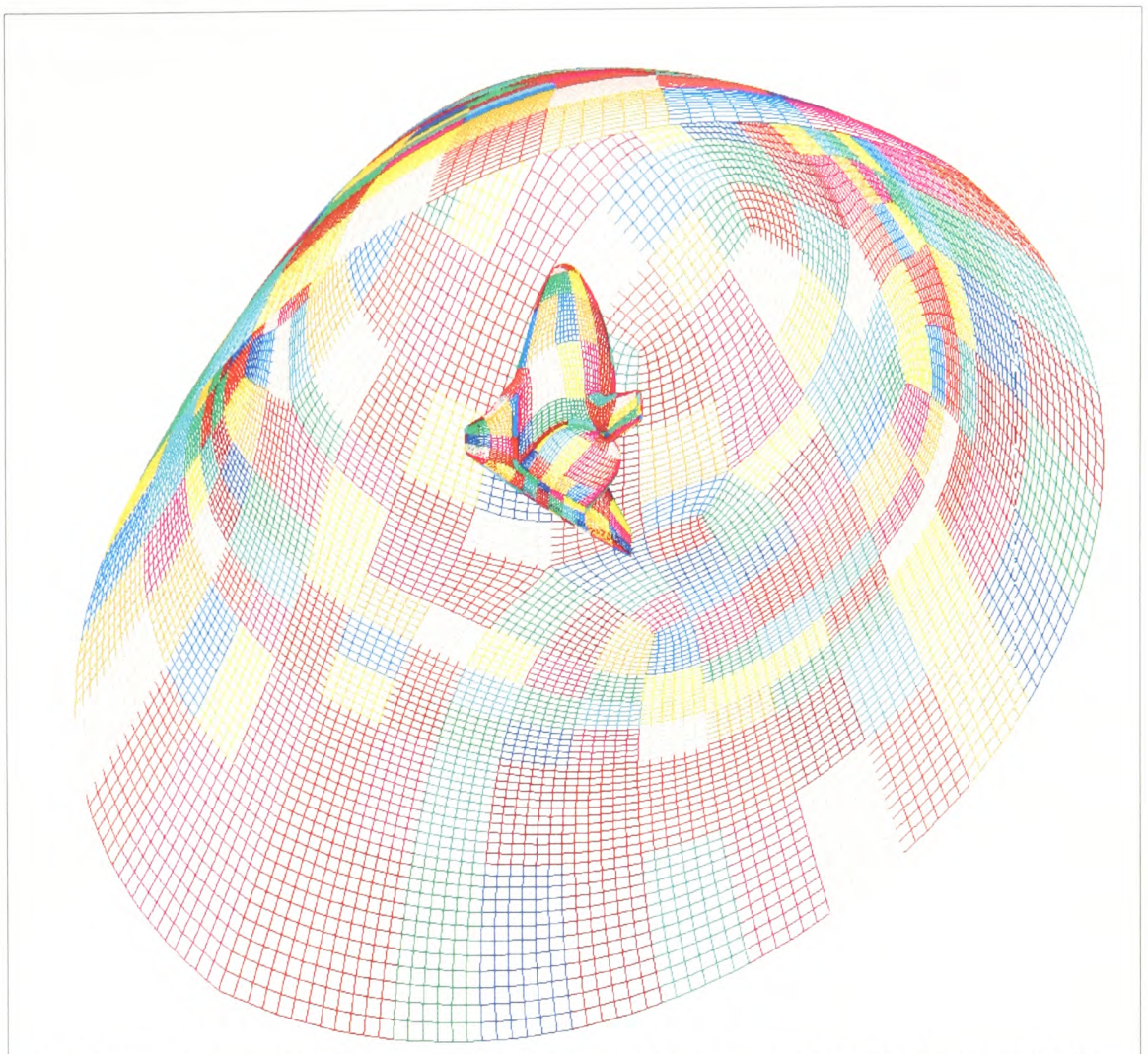


Illustration 1.1.1: GridPro grid of the European Experimental Test Vehicle (EXTV), a structured grid with 780 blocks. Generated by GridPro™

Today Computational Fluid Dynamics (CFD) and related computer simulations are used in many areas of research and development. In these days computer simulation is the enabling technology in the design of vehicles (aeroplanes, cars, ships) as well as combustion engines or turbines to optimise their performance and making them more energy-efficient. Space flight vehicles in particular, are subject to extreme heating rates during reentry from outer space because of intense atmospheric friction, thus requiring a carefully designed thermal protection system. In order to determine those surfaces of a vehicle that are exposed to most severe heat flux along the reentry trajectory at a given angle of attack, extensive simulations need to be performed.

In medical applications, for example (among numerous other cases), to aid doctors before actually performing surgery, simulations of pressure ratios in an artery are now being carried out routinely.

State of the art development of electronic high tech components are not practicable without simulations of electric fields and quantum-mechanical effects.

Interdisciplinary simulations, for example the coupling of the traditional fluid dynamics with electromagnetic fields (magnetohydrodynamics, MHD), examine the influence of electromagnetic fields on an existing flow field to perhaps substitute in future aerodynamic control surfaces.

To meet the demands of modern computational flow simulations, an engineer or software developer simulating a flow field (solver) encounters many difficulties. Traditional CFD grids for aerospace and automotive industries are getting more and more complex, and grids with tens of

millions of points are not uncommon anymore. Completely different types of geometric shapes introduce additional requirements, for instance, in CFD for medical applications. Chemical reactions and electromagnetic fields have an influence on the requirements of the simulation and therefore a need for multiphysics solvers exists.

It is also difficult for a solver developer to perform rapid prototyping because of the missing software development infrastructure, despite the fact that existing libraries can be reused. However, since Moore's law is still intact, namely that every 18 months the overall power of computer systems doubles, new computer systems with new operating systems are constantly becoming available, and therefore quite often major differences exist between these systems causing large efforts even for minor changes. Hence the developer must always readjust his software library to ensure that it still works with new system configurations. This task can consume a lot of time of the actual development process for a solver.

Illustration 1.1.2: A structured grid of a turbine. Generated by GridPro™

This is one of the main reasons why the creation of a simulation software framework for high performance computation and communication using the Java™ runtime environment should be of great importance.

Due to Java's independence from processors and operating systems, any compiled code can be executed without any changes on every platform provides a Java runtime environment, thus guaranteeing complete software portability.

Huge numerical grids need enormous computing resources and if the simulation program performs computations in the multiphysics regime, the computational demand grows dramatically. The only way in getting results for such problems lies in the use of multiple processors. This strategy requires the solution of the essential problem of load balancing that is distributing the total computational load onto the set of processors in order to achieve uniform loads. In order to

optimise parallel speedup an almost uniform computational load is necessary. The difference between an unbalanced versus a balanced system may result in highly different speedup factors. A worst case scenario would be if one processor was running 100% of the load and all others were idle. The most popular software libraries to ensure the communication between the processors are MPI (Message Passing Interface) and PVM (Parallel Virtual Machines). These vendor specific C or Fortran libraries are optimised for MPP (Massive Parallel Processors) systems. MPI/PVM are providing a high performance communication layer between processors. The major disadvantage of MPP systems are to get reasonable dynamic load balancing. In order to achieve this, one has to use complex technologies like domain decomposition and the grouping of messages to obtain dynamic load balancing etc.

1.2 Objective and Scope of the Thesis

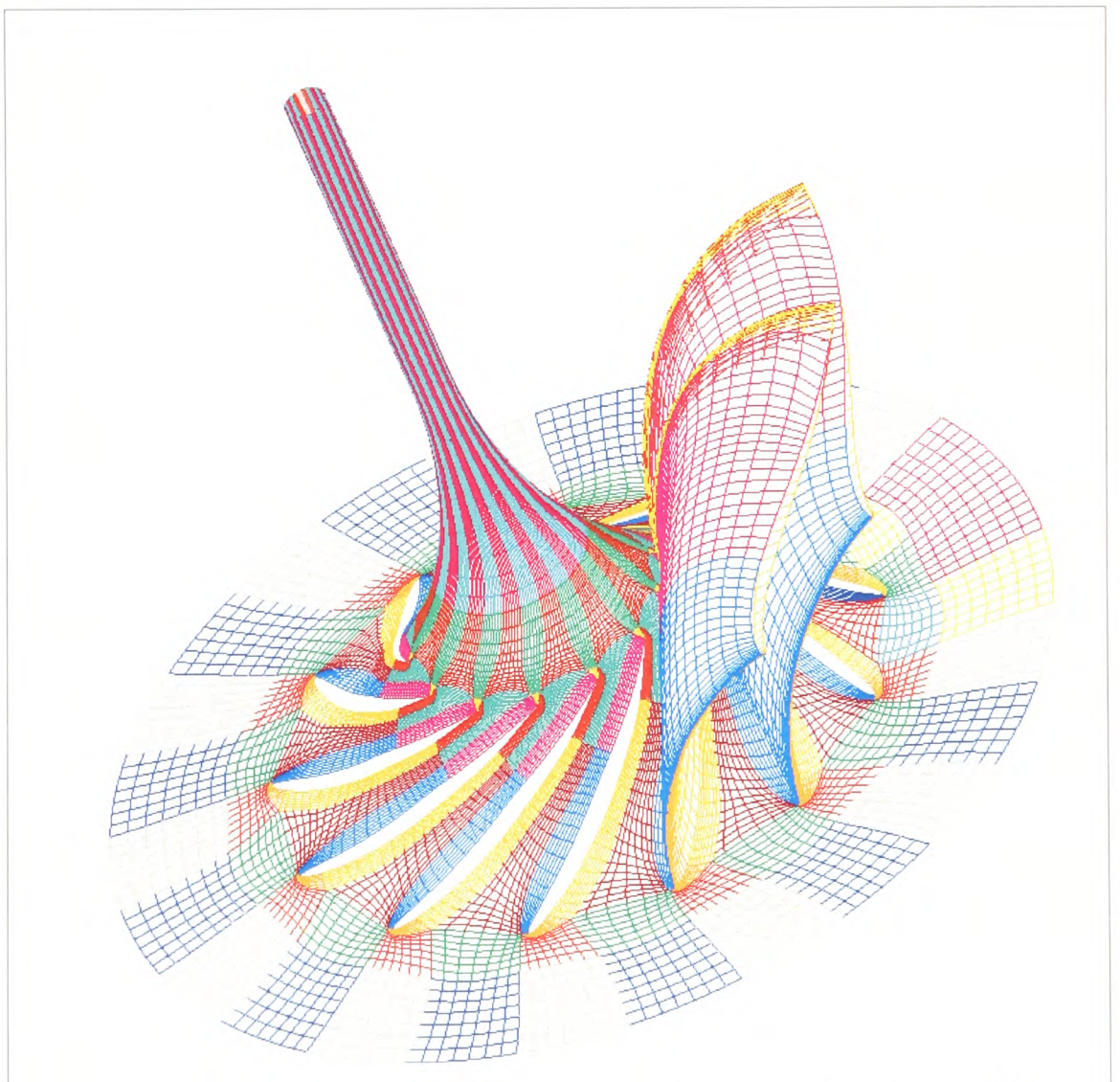


Illustration 1.2.1: Internal view of the turbine shown in Illustration 1.1.2. Generated by GridPro™

However a novel trend has been set up in computer industry, by providing processors with more than one execution core. Two cores on a single processor chip are now common and four core chips will soon become available from Intel and AMD (spring 2007). Other processors like the Niagara 1 (UltraSPARC T1) from Sun Microsystems have 8 cores with 4 execution engines on a single chip that means there are 32 CPUs available per chip. The Niagara 2 available since 2007, has 8 cores with 8 execution engines. UltraSPARC T3 (Rainbow Falls) is announced for 2010 is supposed to have 16 cores with 8 execution engines pre core.

One execution engine is called strand by hardware designers to separate them from software threads. Strands and hardware threads are the same. This line of parallel development demands a completely different parallel strategy that lies in the extensive usage of threads (see chapter 2.5.1

on page 34). A thread of context wastes much less resources than MPI and PVM libraries. In fact it demands substantially less resources than a process handled by the operating system. Therefore a multithreaded application is able to run thousands of threads in a single process. Thread programming for HPCC and how it is possible to achieve excellent parallel efficiency, with the new developed Direct-Neighbour-Synchronization (DNS) will be discussed in chapter 2.5 on page 34. It will be shown in chapter 8.1.1.3 on page 128 that threads give excellent dynamic load balancing.

In addition, Java has great advantages over languages like C, C++ and Fortran because the thread concept is a built in feature in the language, and hence there is no need to link against libraries that are operating system dependent.

In addition to the extensive usage of threads for high performance computing and communication in a pure Java runtime environment, the software framework created as described in this work provides even more important features.

There exist loaders and writers for various 3D file formats that free the programmer from dealing with complex geometries. In order to reduces geometry complexity boundary fitted coordinates are utilized, performing a transformation from physical space to computational space. In computational space the complex geometry is represented by a uniform rectangular multiblock grid. Naturally the physical equations have to be transformed as well but there type does not change.

All communication between processors for dynamic load balancing is done by the framework itself. The solver developer can therefore concentrate on the solution of the governing equations on a simply connected domain also called a block.

A user of the framework can exchange the default solver by his own solver version during the compute session, sending the Java compiled byte code to the compute host.

Due to the implemented client/server concept, the Internet capabilities and the interactive steering features of the framework the possibility of collaborative engineering is provided over an encrypted secure connection.

In summary the ***JUSTGRID*** framework takes care of all geometrical complexity, which is one of the most difficult part in three dimensional simulations, and provides complete static as well as dynamic load balancing.

2 High Performance Computation Fundamentals

2.1 Parallel System Models

2.1.1 Massive Parallel Processing System (MPP)

MPP systems or so called Beowulf cluster systems sharing there high speed interfaces (crossbar) or network interfaces only. These are computer farms with many single computers in a rack or many single system boards in one computer case.

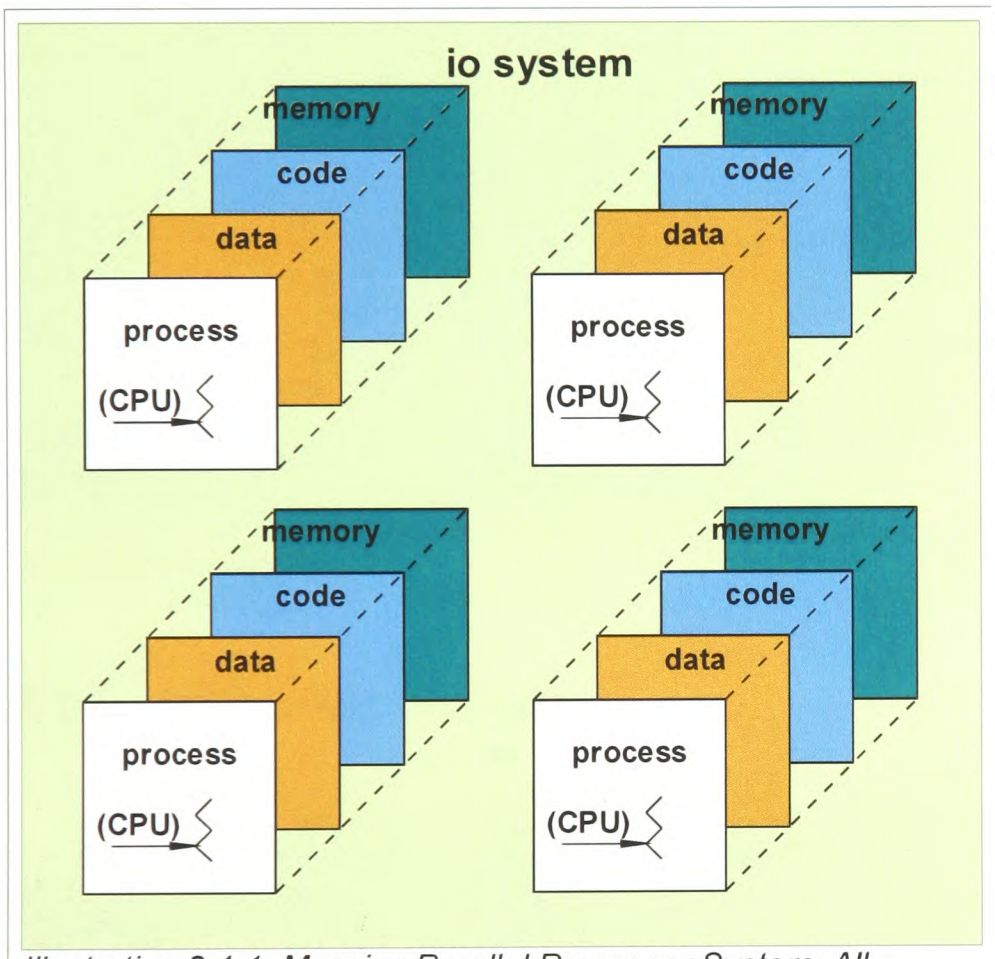


Illustration 2.1.1: Massive Parallel Processor System. All computing nodes are sharing one communication layer.

The Beowulf type cluster systems are very popular because they are significant cheaper than a huge SMP machine. One of the serious disadvantages of MPP systems is the dynamic load balancing, one has to do a lot of work and thinking about complex strategies to get a balanced processor load.

2.1.2 Symmetric MultiProcessing System (SMP)

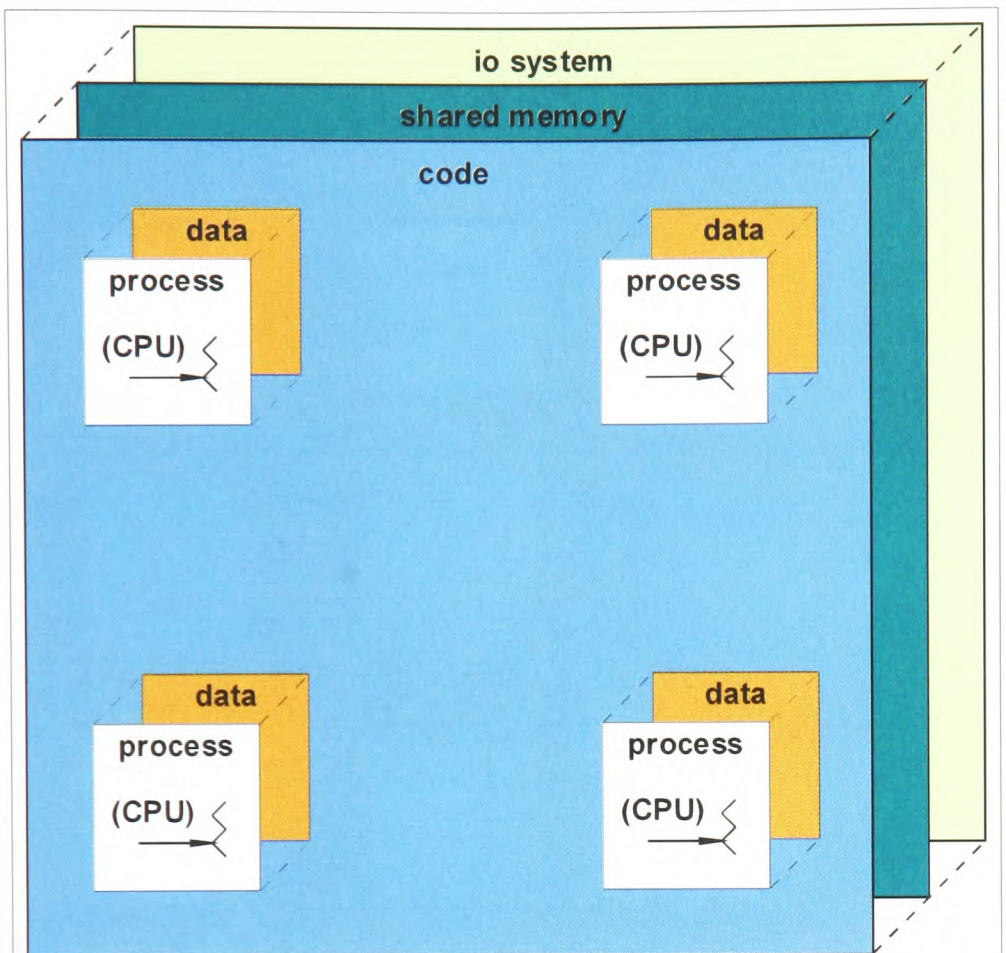


Illustration 2.1.2: Symmetric Multi-Processor System (SMP) architecture diagram.

In SMP systems the parallel computation shares the IO subsystem a defined amount of memory and the program code is the same for every processor. Only the data area in the memory is allocated to the processors.

2.1.3 SMP with Threads

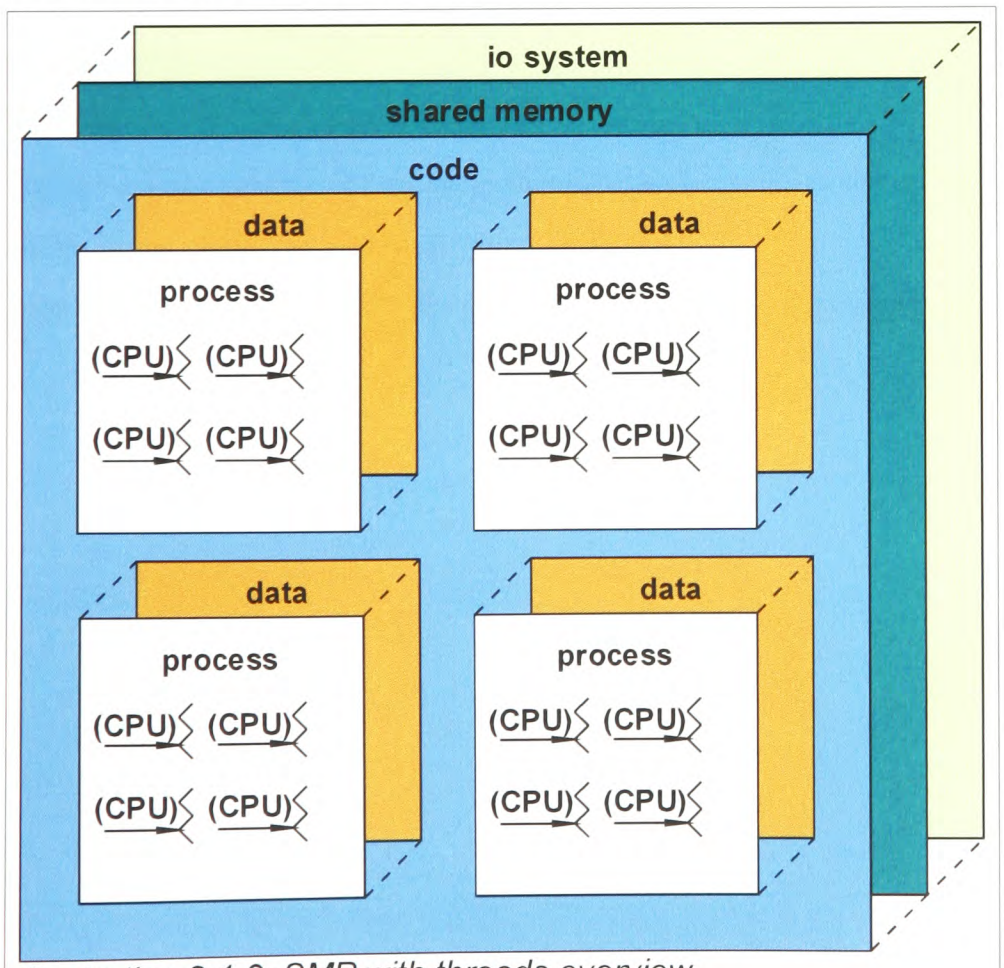


Illustration 2.1.3: SMP with threads overview

SMP with threads is going one step further that is everything is shared be it data, code, or io by the execution contexts. (Threads)

2.1.4 One multithreaded process

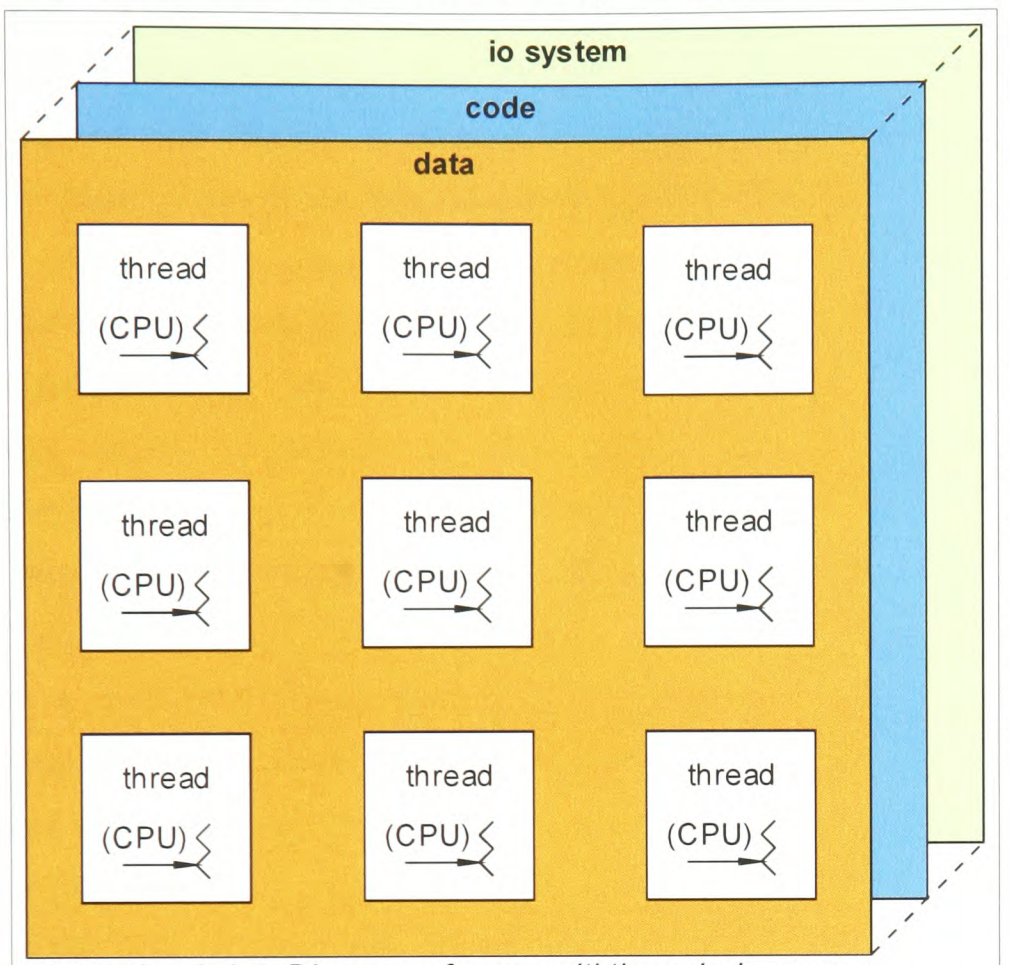


Illustration 2.1.4: Diagram of one multi threaded process

From the developers view this model is the easiest one. With a modern operating system like Sun Microsystems Solaris the compute session gets the dynamic load balancing for free if enough parallel threads are started. The difficulty in the threaded programming model is the synchronization between threads and the way to get exclusive access on specific data. Java has solutions for these problems built in.

2.2 Legacy Codes

The newer versions of Java are serious competitors to the traditional HPCC programming languages like FORTRAN or C/C++. The single-processor performance of a Java code is now on par with C++, and the speedup on common symmetric multi processor (SMP) machines is excellent.

Runtime (2GHz, Pentium4, 1GB Memory)	time in s
Sun JVM 1.4.2_02 (-server)	2.12
GNU gcc version 3.3.1 (-O3 -mcpu=pentium4)	3.16

Table 2.2.1: Sequential matrix multiplication using a 30 times 30 matrix doing 10000 iterations on a Linux Pentium 4 PC

2.3 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.

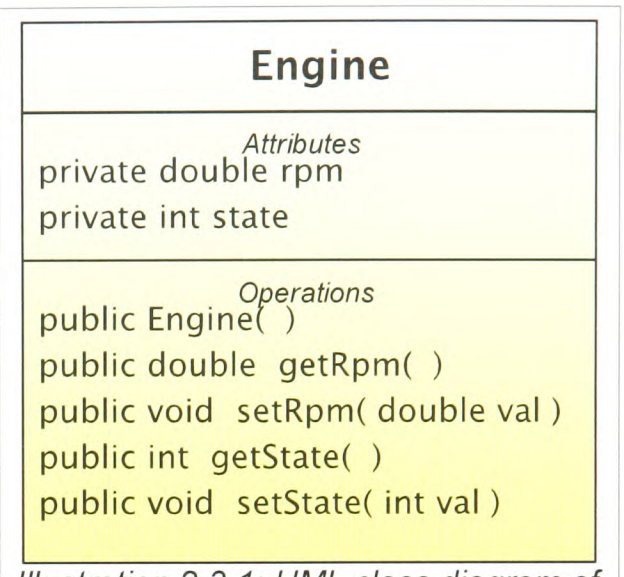


Illustration 2.3.1: UML class diagram of a sample engine class

2.4 Java

2.4.1 Java Technologies

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

2.4.2 Object Oriented Programming

See: Object Oriented Programming in chapter 2.3 on page 32.

2.4.3 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.

2.4.4 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.

2.4.5 Portability

Most languages are compiled directly to the machine code of the machine on which they are to be 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 byte code in native machine code.

2.4.6 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.

2.4.7 Multithreading

In Java, concurrency is achieved via the thread (see also chapter 2.5 on page 34) 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 its 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.

Threads are a substantial part of the Java programming language. Java is the only general programming language that does not need external libraries for parallel programming, because everything needed is built into the language.

2.4.8 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.

2.4.9 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.

2.5 Thread programming in HPCC

This chapter gives a general introduction about threads and shows the special issues in thread programming for high performance computing and communication. It also demonstrates how it is possible to achieve excellent parallel efficiency utilizing the newly developed *Direct-Neighbour-Synchronization (DNS)*, while dealing with tens of thousands or more threads.

The thread concept as basic parallelization strategy, delivers an enormous number of options to speed up parallelization, since fine tuning by threads on all levels of parallelization (i.e, domain decomposition, numerical algorithm, loops etc.) of a computation is possible

2.5.1 What are Threads?

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 **selfscheduling**. Java also provides a mechanism for **synchronizing** threads and for **sending messages** between threads.

Multithreaded programs extend the idea of multitasking one level further such that individual programs (processes) will appear to perform multiple tasks at the same time. Each task is called a *thread* which is the short form for thread of control. Programs that can run more than one thread at a time are said to be *multithreaded*. A thread consist of three parts : a virtual CPU, the code to be executed and the data the code works on.

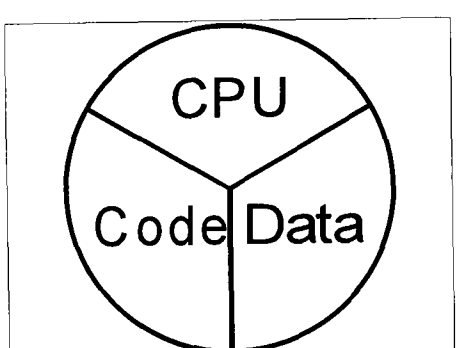


Illustration 2.5.1: *A thread or execution context*

2.5.2 Threads vs. Processes

The architectural differences between threads and processes are shown in chapter 2.1 on page 29. The results shown in chapter 8.1.1.3 on page 128 demonstrates perfectly the lightweight character of threads.

2.5.3 Models of Thread implementations

Three different types of thread implementations are available in various Java Virtual Machines (JVM).

2.5.3.1 Many-to-One (Green Threads) Thread Model

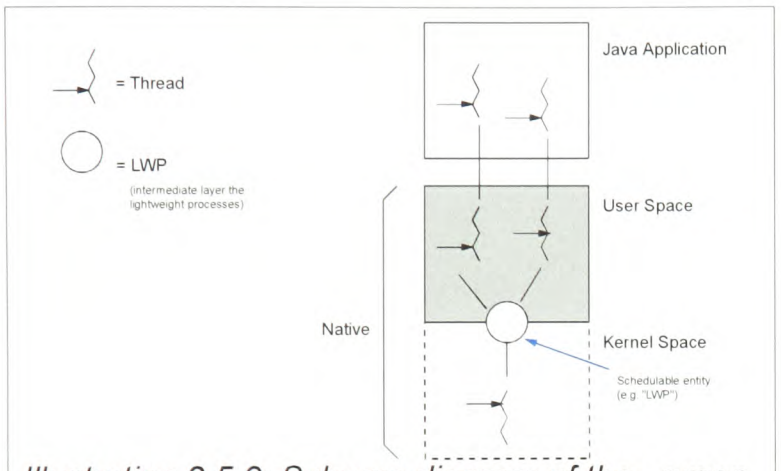


Illustration 2.5.2: Schema diagram of the ,,green thread" model

The Many-to-One model was the first thread model Java implements. Nowadays it is only used with Java embedded Virtual Machines. This model puts all Java application threads to one native kernel thread. It is **not** possible to build a multiprocessor application with this model.

2.5.3.2 One-to-One Thread Model

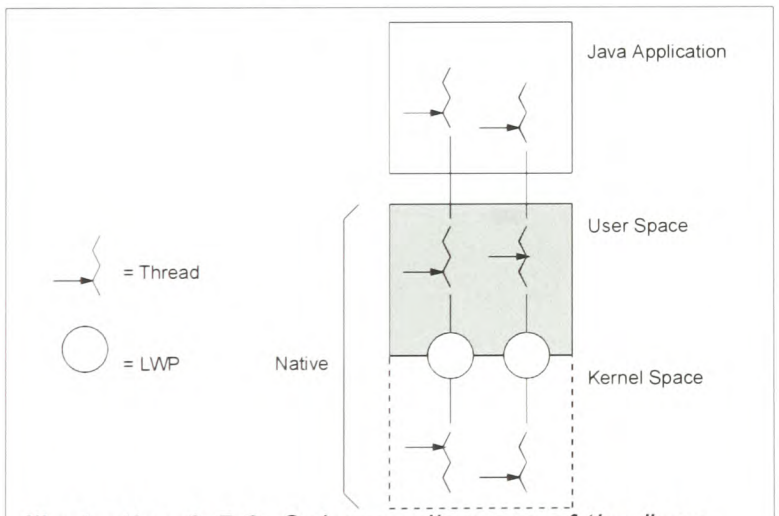


Illustration 2.5.3: Schema diagram of the "one-to-one" thread model. One application thread is mapped to one kernel thread.

The One-to-One model puts one application thread on one kernel thread. It uses more kernel resources than the Many-to-Many model but for SMP machines with a large number or processors the thread context switch is much faster and performance increases.

2.5.3.3 Many-to-Many Thread Model

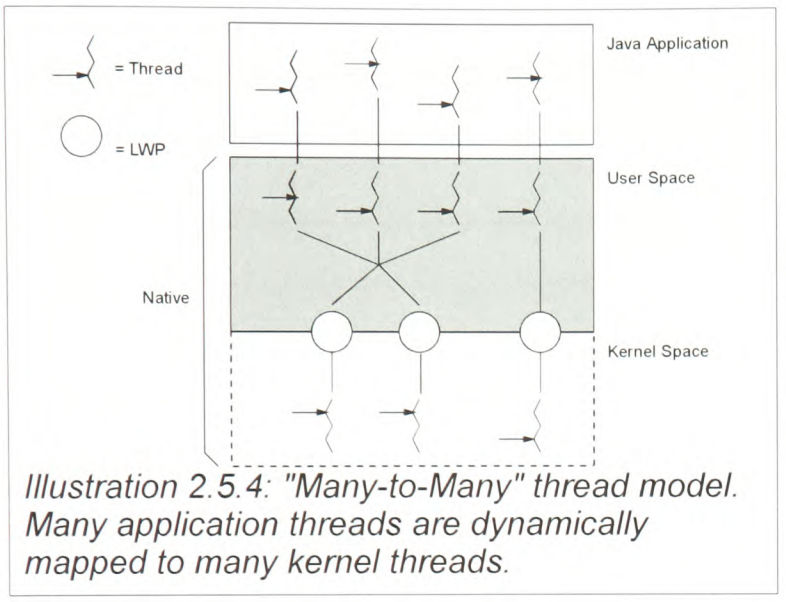


Illustration 2.5.4: "Many-to-Many" thread model. Many application threads are dynamically mapped to many kernel threads.

The Many-to-Many Model was for a long time preferred over the One-to-One model because of much lesser use of kernel resources. Moreover on a dual processor system it was the best model to run thousands of threads on a single machine. But now the focus is more on performance and memory is cheap these days, so the Many-to-Many model is no longer in use.

2.5.3.4 Best Thread Model for JUSTGRID

Today most modern operating systems like Linux, Mac OS X, Solaris, and Windows are utilising the One-to-One Thread Model to support the optimal performance on multi-processors systems. Furthermore, this also is the best mechanism for supporting the ***JUSTGRID*** framework.

2.5.4 Thread Scheduling

A Thread object can exist in many different states throughout its lifetime. Illustration 2.5.5 Shows this idea:

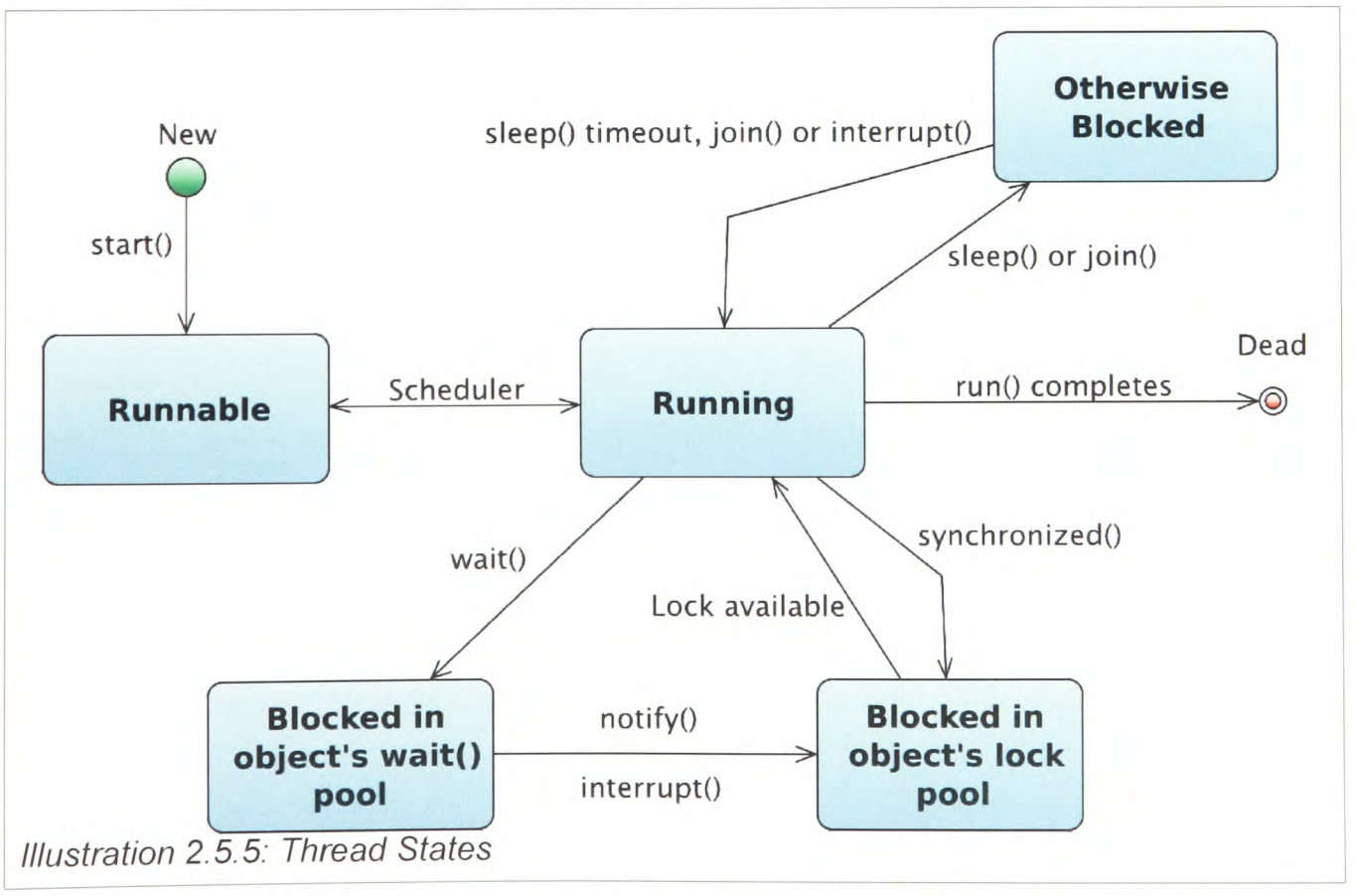


Illustration 2.5.5: Thread States

With the Java Development Kit (JDK) version 1.2, the `suspend()`, `resume()`, and `stop()`

methods have been deprecated. `suspend()` is a deadlock prone and `stop()` is unsafe in terms of date protection.

Although the thread becomes runnable, it does not necessarily start running immediately. Only one action at a time can be done on a machine that has a single processor.

In Java technology, threads are usually *preemptive*, but not necessarily timesliced (the process of giving each thread an equal amount of CPU time). It is a common mistake to believe that "preemptive" is a fancy word for "does timeslicing". The behaviour of most JVM implementations appears to be strictly preemptive. But across JVM implementations, there is no guarantee of preemption or timeslicing. The only guarantees lie in the developer's use of `wait()` and `sleep()`. The model of a preemptive scheduler is that many threads might be runnable, but only one thread is actually running. This thread continues to run until either is ceases to be runnable, or another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *preempted* by the thread of higher priority, which gets a chance to run instead.

A thread might cease to be runnable for a variety of reasons. The thread's code can execute a `Thread.sleep()` call, deliberately asking the thread to pause for a fixed period of time. The thread might have to wait to access a resource, and cannot continue until that resource becomes available.

All threads that are runnable are kept in pools according to priority. When a blocked thread becomes runnable, it is placed back into the appropriate runnable pool. Threads from the highest priority non-empty pool are given CPU time.

2.5.5 Thread Synchronization

2.5.5.1 The Problem

Imagine two threads having a reference to a single instance of a stack class. One thread is pushing data onto the stack and the other one, more or less independently, is popping data off the stack. In principle, the data is added and removed successfully. However, there is a potential problem.

Suppose thread *a* is adding and thread *b* is removing characters. Thread *a* has just deposited a character, but has not yet increment the character index counter. For some reason this thread is now preempted. At this point, the data model represented in the stack object is inconsistent.

2.5.5.2 *The Object Lock Flag*

In Java technology, every object has a flag associated with it. One can think of this flag as a "lock flag". The keyword `synchronized` enables interaction with this flag, and allows exclusive access to code that affects shared data. The following sample shows two "synchronized" methods of our stack implementation mentioned in 2.5.5.1.

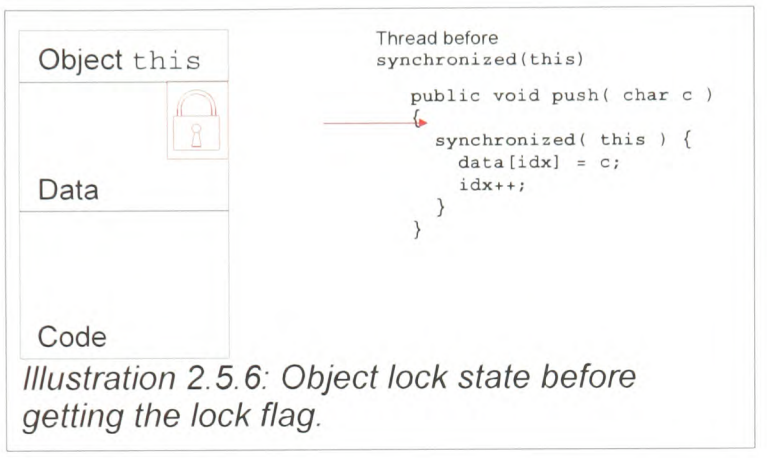


Illustration 2.5.6: Object lock state before getting the lock flag.

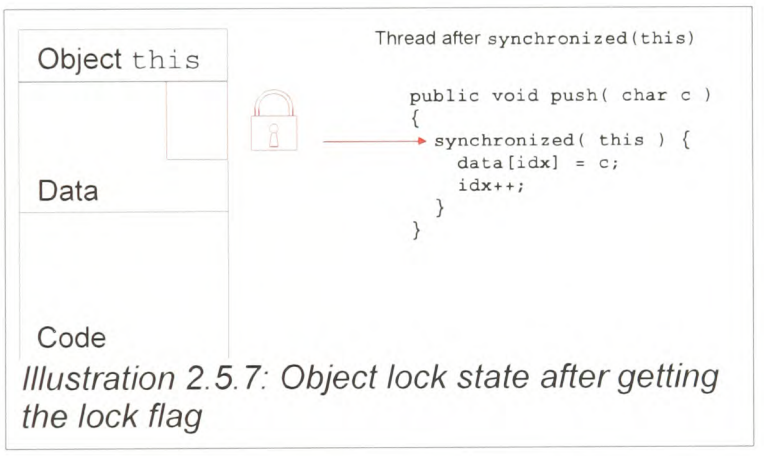


Illustration 2.5.7: Object lock state after getting the lock flag

The thread waiting for the lock flag of an object cannot resume running until the flag is available. Therefore, it is important for the holding thread to return the flag when it is no longer needed.

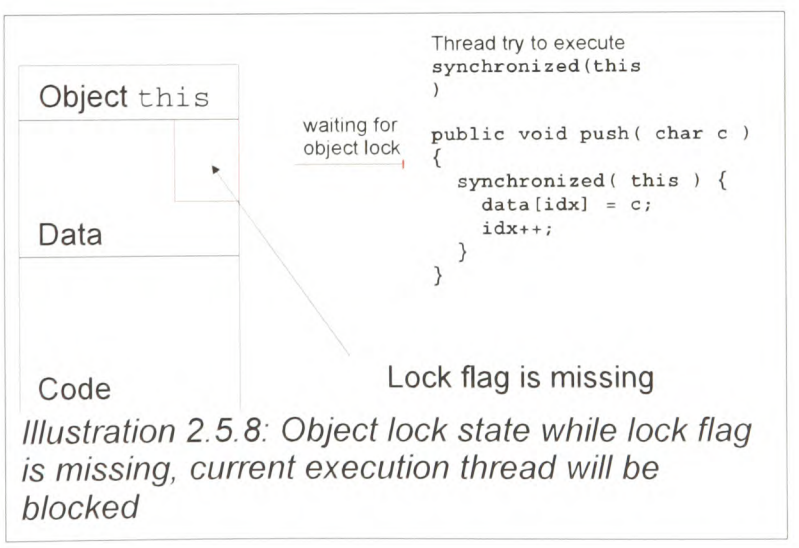


Illustration 2.5.8: Object lock state while lock flag is missing, current execution thread will be blocked

2.5.6 Thread programming challenges.

How to deal with tens of thousands of threads in one simulation process? The answer to this question is very easy if the problem to be solved can be divided into autonomous parts without any communication between theses parts. Just start one thread per part or so called domain or block and wait until they all become ready. But for all of the problems we have in mind (CFD, MHD, ...) communication between blocks (domains) is crucial in order to solve the problem at all. Communication between threads (blocks) cannot be done at arbitrary times during the computation. It has to be done at particularly defined points to be sure to transport valid data only between the blocks. Generally this has to be done before a numerical iteration step is initiated. Hence the difficulty is to find an efficient mechanism for threads to wait for other threads to become ready. As shown in chapter 2.5.5 the Java environment provides a synchronization mechanism for threads , but there are different techniques to achieve the best possible parallel efficiency.

2.5.6.1 Comparison of Thread synchronization techniques within a HPCC code

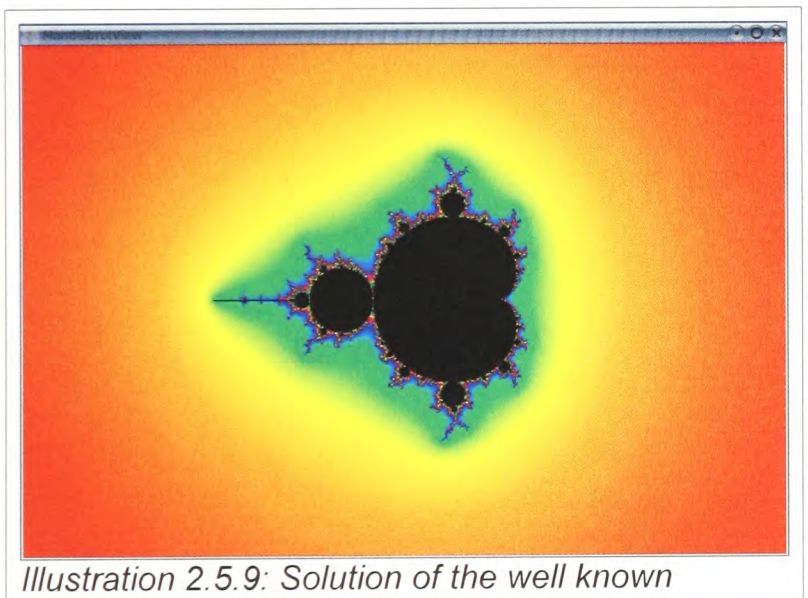


Illustration 2.5.9: Solution of the well known Mandelbrot Set.

A Solver for the well known Mandelbrot Set was developed as a test case for three reasons. First, from the numerical aspect the Mandelbrot Set problem is highly load imbalanced over the complete solution process. This is important to see the different behaviour of the compute threads (blocks). Second is the fact that there is no data exchange between the blocks, otherwise it would be impossible to implement a test case without any synchronization. The third and last reason was the possibility to visualize the three different synchronization methods to depicting the dead lock and race condition errors.

All three synchronization methods use 64 threads. One compute thread is responsible for one block (stripe).

The Mandelbrot Set is a fractal named named after Benoît Mandelbrot. Greatly illustrated and described in “The Beauty of Fractals. Images of Complex Dynamical Systems” by Heinz-Otto Peitgen and Peter H. Richter [HPPR].

2.5.6.2 No synchronization

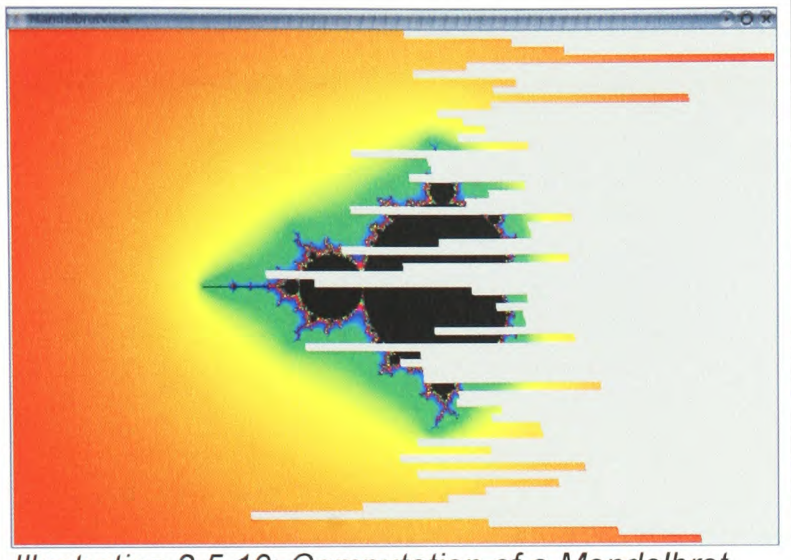


Illustration 2.5.10: Computation of a Mandelbrot Set without any synchronization between the compute threads.(Snapshot during computation)

This sample shows the numerically imbalanced character of the Mandelbrot Set computation. While one thread has already finished other threads are many iterations behind. The maximum iteration number is also the possible maximum iteration difference if there is no synchronization. Because of the lack of any synchronization this is the fastest method for any number of processors.

2.5.6.3 Global synchronization

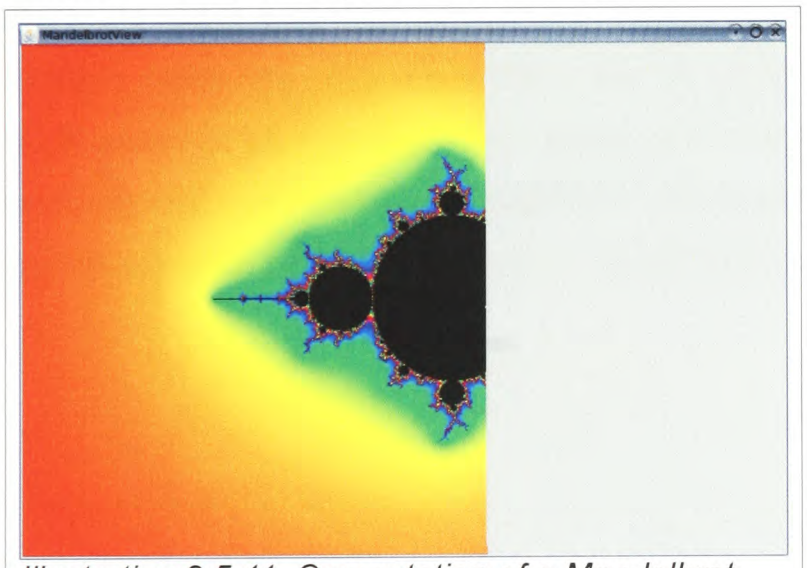


Illustration 2.5.11: Computation of a Mandelbrot Set with global synchronization between the compute threads (Snapshot during computation)

To achieve global synchronization all threads need to be synchronized with one global object lock flag. Maximum iteration difference over all domains is 1. It takes a few lines of code only to implement this synchronization method, and the performance efficiency lost for one and two processors compared with the "no synchronization" method is really small. With more than two processors this synchronization method becomes more and more inefficient because of the rising number of native system threads, waiting for the last thread to become ready. See the result in 2.6.3 on page 46.

2.5.7 Race condition and deadlock - common programming pitfalls in parallel execution systems

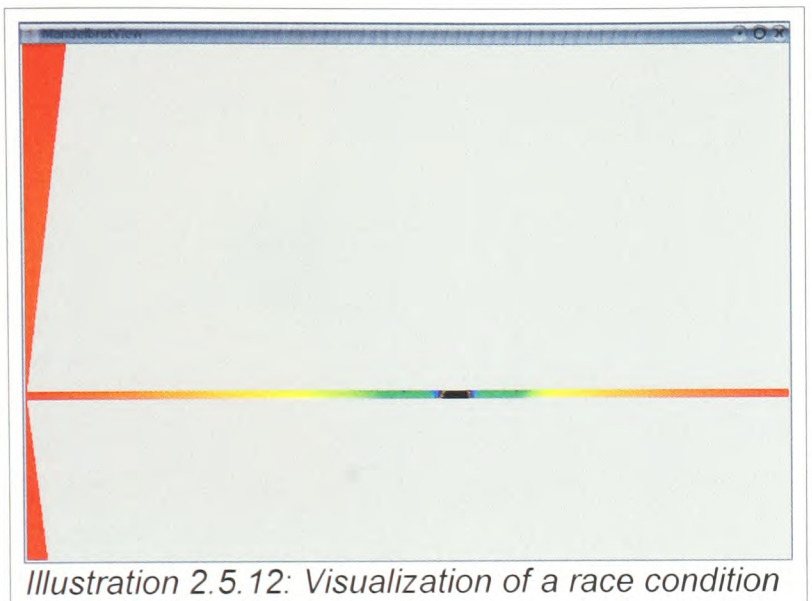


Illustration 2.5.12: Visualization of a race condition error

The two most common pitfalls in parallel programming with synchronization are the *race condition* and the *deadlock*. The "race condition" error occurs if a thread suddenly is no longer synchronized and simply keeps running (see Illustration beside). Next, suppose thread (A) waits for thread (B) to become ready and thread (B) waits for thread (A) to become ready and together they wait forever, this is called "dead lock" error. The basic approach avoiding "race conditions" is serializing the problem which, however, is no option for ***JUSTGRID***. Even the known solutions for avoiding "deadlocks" like the Banker's algorithm (using maxima values as a break condition) or the wait/die algorithm one threads waits until the other dies, will not give a consistent valid solution. The only way to avoid such errors is careful algorithm design and more comprehensive testing of the complete simulation environment. In fact the ***JUSTGRID*** framework is also responsible for synchronization and communication between threads a developer does not need to take care about these difficult tasks.

2.6 *Direct Neighbour synchronization (DNS)*

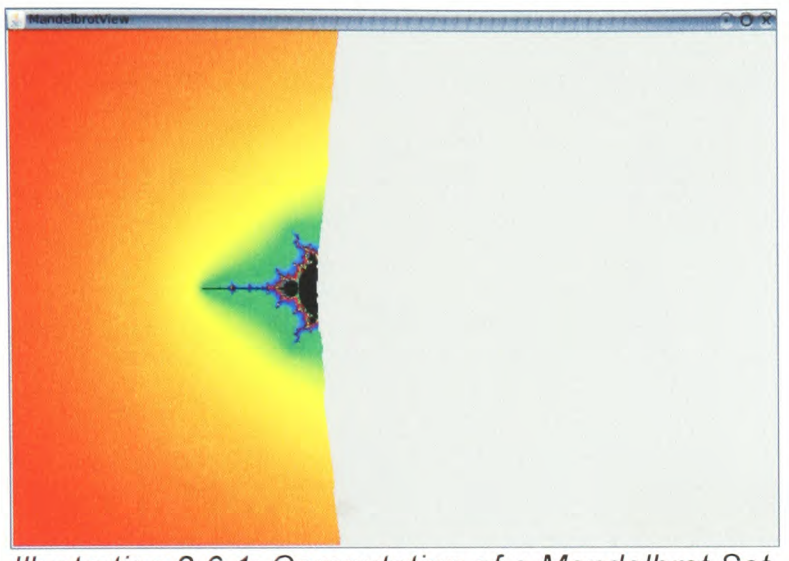


Illustration 2.6.1: Computation of a Mandelbrot Set with Direct Neighbour synchronization between the compute threads

To reduce the number of idle threads the novel technique of **Direct Neighbour synchronization (DNS)** was developed. Threads are synchronized with direct neighbours only, one object lock flag per thread. Thus the maximum iteration difference between neighbours was reduced to 1. In Illustration 2.6.1 one can see the centre blocks with the most numerical load (colour=black) are behind the blocks at the boundary. This synchronization technique gives the operating system much better handle to distribute the compute threads over all available native system threads. For the DNS an additional Object for the synchronization is needed to avoid the deadlock prone.

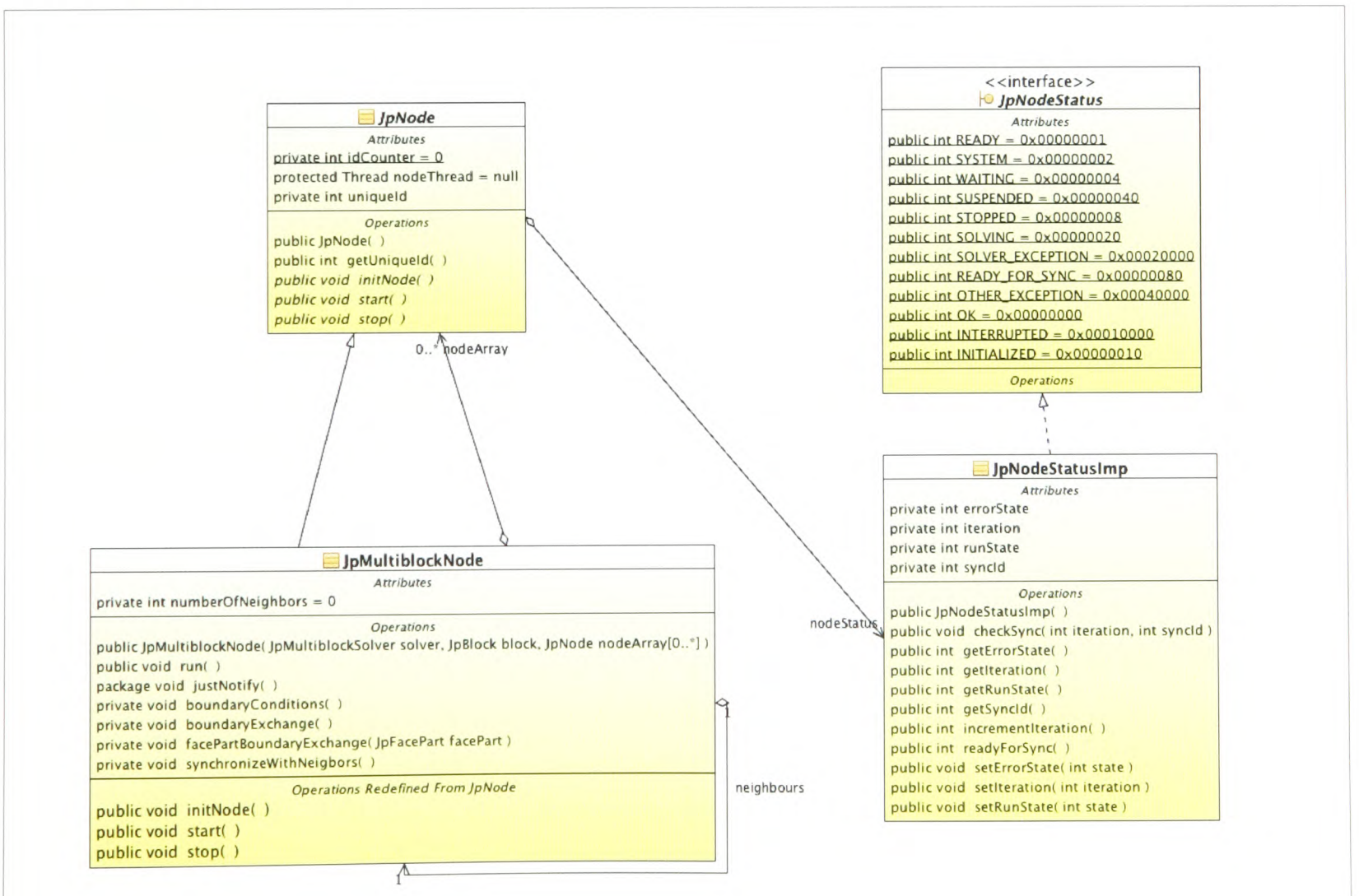


Illustration 2.6.2: UML Class diagram for a JpMultiblockNode with Direct Neighbour Synchronization implemented with JpNodeStatusImp

The class JpNodeStatusImp in JUSTGrid implements the DNS. Every compute node (thread), in

our case JpMultiblockNode, is initialized with an Array of references to its neighbours and contains one instance of JpNodeStatusImp. Additionally JpNodeStatus indicates the current status of a Node.

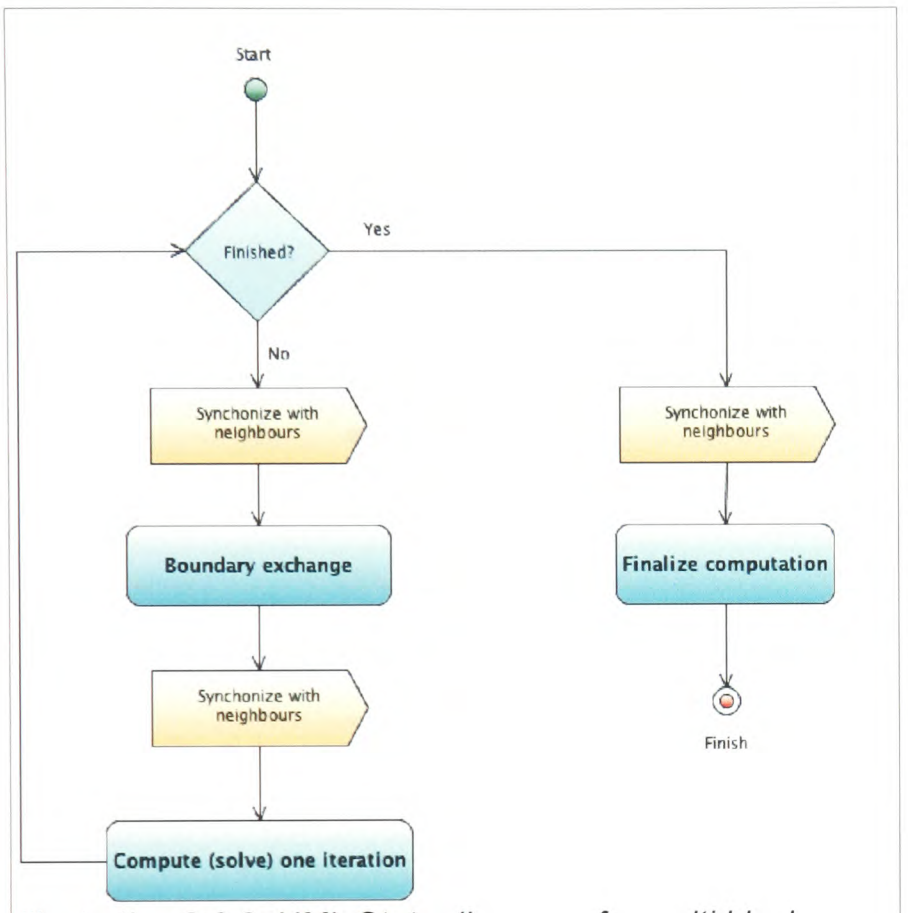


Illustration 2.6.3: UML State diagram of a multi block compute node.

Due to the fact that there are normally more than one synchronization points in a computation cycle (iteration) the DNS status object must recognize the different synchronization points (Illustration 2.6.3). The current iteration number is insufficient to identify a unique point in the execution path. Therefore it was necessary to extend the "`checkSync`" method call by an additional identification parameter "`syncId`".

2.6.1 JpMultiblockNode

Every instance of JpMultiBlockNode represents one execution thread. On every synchronization point all neighbour states must be checked. If one ore more neighbours are not ready to synchronize the current thread go into "wait"-state.

```
private void synchronizeWithNeigbors() throws InterruptedException
{
  int iteration = nodeStatus.getIteration();
  int syncId = nodeStatus.readyForSync();

  if (neighborNode != null)
  {
    for (int i = 0; i < neighborNode.length; i++)
    {
      if (neighborNode[i] != null)
      {
        neighborNode[i].nodeStatus.checkSync(iteration, syncId);
      }
    }
  }
}
```

To check all neighbours first the current iteration and the current syncId will be received from the own status object (JpNodeStatusImp). Then all neighbours will be checked for their state.

2.6.2 JpNodeStatusImp

The method "`readForSync`" has two tasks providing the caller method with the current syncId and notifying all waiting neighbours (threads) about the new state of this node.

```
synchronized public int readyForSync()
{
  this.syncId++;
  this.setRunState(READY_FOR_SYNC);
  notifyAll();

  return syncId;
}


synchronized public void checkSync(int iteration, int syncId)
{
  while (true)
  {
    if (this.iteration > iteration)
    {
      break;
    }

    if ((this.iteration == iteration) && (this.syncId >= syncId))
    {
      break;
    }

    try
    {
      wait();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }
}
```

If the current compute node is behind or exactly at the same state as the comparing neighbour node the execution thread will immediately return from the "`checkSync`" method. If the current node is ahead compared to the neighbour node the execution thread will be set into the wait-state. The execution thread will wait at this point until it receives the notification from the "`readyForSync`" method.

2.6.3 Efficiency results for the different synchronization methods

The "No Synchronization" (NS) test case is the reference for the "Global Synchronization" (GS) and the "Direct Neighbour Synchronization" (DNS).

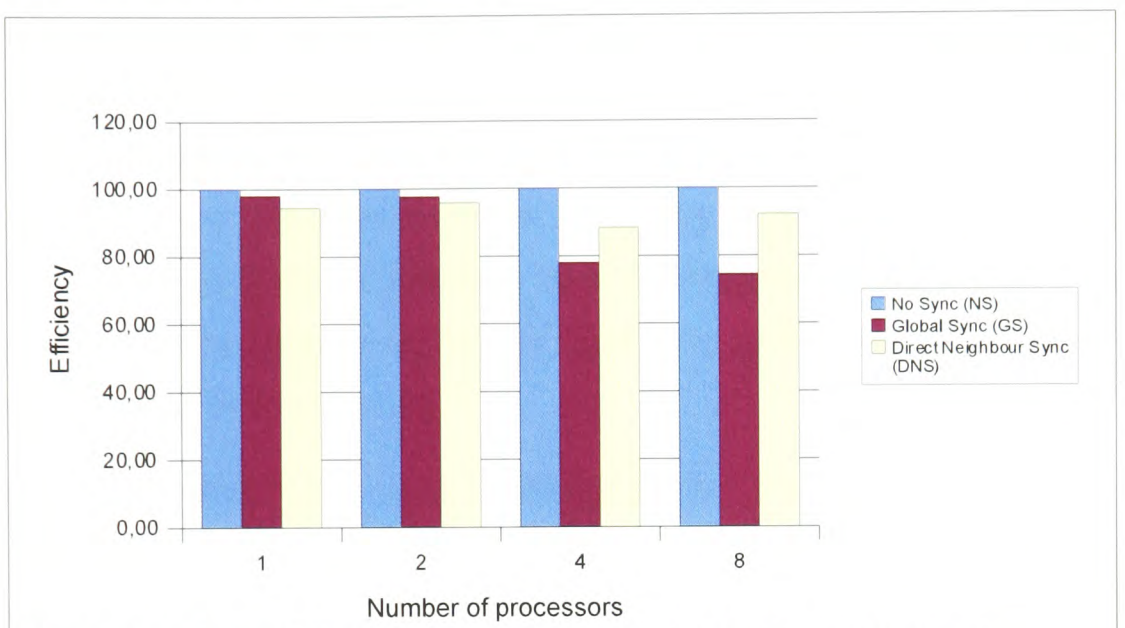


Illustration 2.6.4: Efficiency results for the different synchronization methods increasing the number of processors

Due to the more complex implementation of the DNS the GS shows better results for 1 and 2 processors. But starting with 4 or more processors the DNS demonstrates its advantage against the GS even for this simple Mandelbrot Set test case. Nowadays the computer industry provides multicore processors with many native threads per core, for instance, there already exists the Sun Microsystems UltraSPARC T2 with 8 cores and 8 threads per core, all in one processor. This processor presents itself to the operating system as a collection of 64 single CPUs accessing one shared memory system. Other companies like Intel have similar multicore systems on their road map. With such systems the "Direct Neighbour Synchronization" is a powerful strategy to achieve efficient load balancing over all available CPUs, independent of their number.

3 Multiphysics Framework - *JUSTGRID*

3.1 Introduction

JUSTGrid is a completely Java based software environment for the user/developer of HPC software. JUSTGrid 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. JUSTGrid 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. JUSTSolver is a pure Java CFD solver plugin for JUSTGrid, based on finite volume technique, and thus can be used for any kind of hyperbolic problem (system of hyperbolic equations).

JUSTGrid 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, ***JUSTGrid*** is completely portable, and can be used on any computer architecture across the Internet ,as long as a Java Runtime Environment (JRE) is provided.

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. The parallelization is entirely based on the Java thread concept. 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.

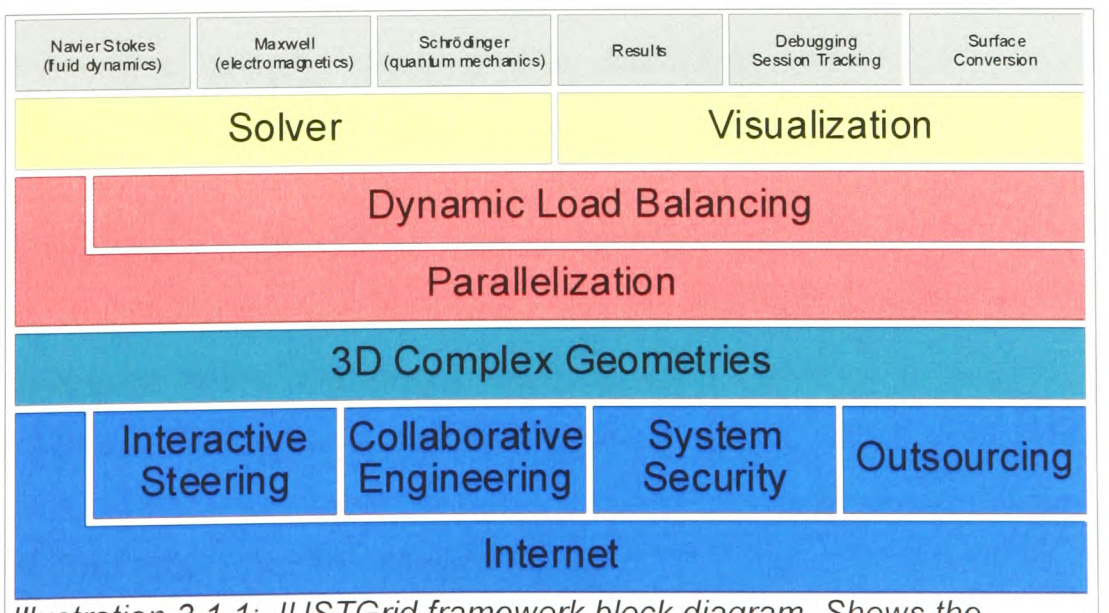


Illustration 3.1.1: JUSTGrid framework block diagram. Shows the different parts of the JUSTGrid architecture.

JUSTGrid provides a layer, the *solver package layer*, to be implemented on the client site. 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 *JavaSolver-Interface* therefore provides the software infrastructure to the the other two layers, and thus is usable for a large class of computational problems based on finite volume formulation. 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.

3.2 Highlights

- **Replace the default solver with your own solver (mathematics).**

  The design of JUSTGrid allows to replace the default computation class (Solver, Cell, SessionHandler, BoundaryHandler, ...) on the server, except the Session and Master Implementation.

- **Free configurable solver plugin service**

  Set/Get any value to your solver (Reflection API).

- **Exchange the solver online during computation.**

  The exchange of a specific class on the **JUSTGrid** server can be initiated while the computation is running without a restart cycle.

- **Dynamic load balancing obtained for free on SMP Architectures.**

  Dealing with multithreaded architectures transfers the responsibility for the load balancing from the application to the operating system. Modern operating systems like Sun Solaris are very efficient in distributing the thread load on the available collection of CPUs.

- **Simple geometrical model for the programmer.**

  JUSTGrid frees the programmer from dealing with complex geometries. The programmer

focuses on a cell only that is in a mathematical universe where every edge has normalized length 1. The transformation from the physical- to the mathematical- coordinate system is done by **JUSTGrid**.

- **Simple Solver API (interface)**

  The motto observed during the whole design process is that of Einstein who said: *Make it as simple as possible but not simpler.* For example, if one likes to write his own multiblock solver one has to implement only a single method named `solve`. For other types of solvers only a few more methods need to be implemented. Illustration 3.1.1 on page 47 depicts an UML class diagram of the **JUSTGrid** solver interfaces.

- **OnlineVisualization on demand**

  JUSTGrid provides access to all computational data in the solution domain at any arbitrary state of the computation. Illustrations 3.5.3 and 3.5.2 are showing online visualizations of the solution domain.

- **Collaborative engineering**

  Via a unique Session-ID, multiple clients are able to connect to the same compute session on the server. As an example: if an engineer wants to ask an expert about the correctness of his computation which is currently running, the engineer sends the Session-ID to the expert, who could then connect to this compute session and visualize the computation online, providing his feedback to the engineer.

- **Multiple sessions on one server.**

  The **JUSTGrid** server is able to run as many sessions as you want; it is only limited by the available resources on the server system.

- **Application and network security**

  Java has a very smart security architecture that protects your code and data from unauthorized access or modification. **JUSTGrid** benefits from these application security features and uses the network security layer for client/server communication.

- **Loaders and writers for structured and unstructured grids and TecPlot™ data files are available.**

  Data files can be stored on the client as well as on the server side.

3 Multiphysics Framework - JUSTGrid

- **Modern object oriented software architecture**

  The object oriented architecture allows to benefit from techniques like inheritance, data encapsulation and message passing. These are some of the features that make the code more robust and maintainable.

- **Automatic topology recognition**

  The framework finds all matching edges and faces.

3.3 Client/Server internet architecture

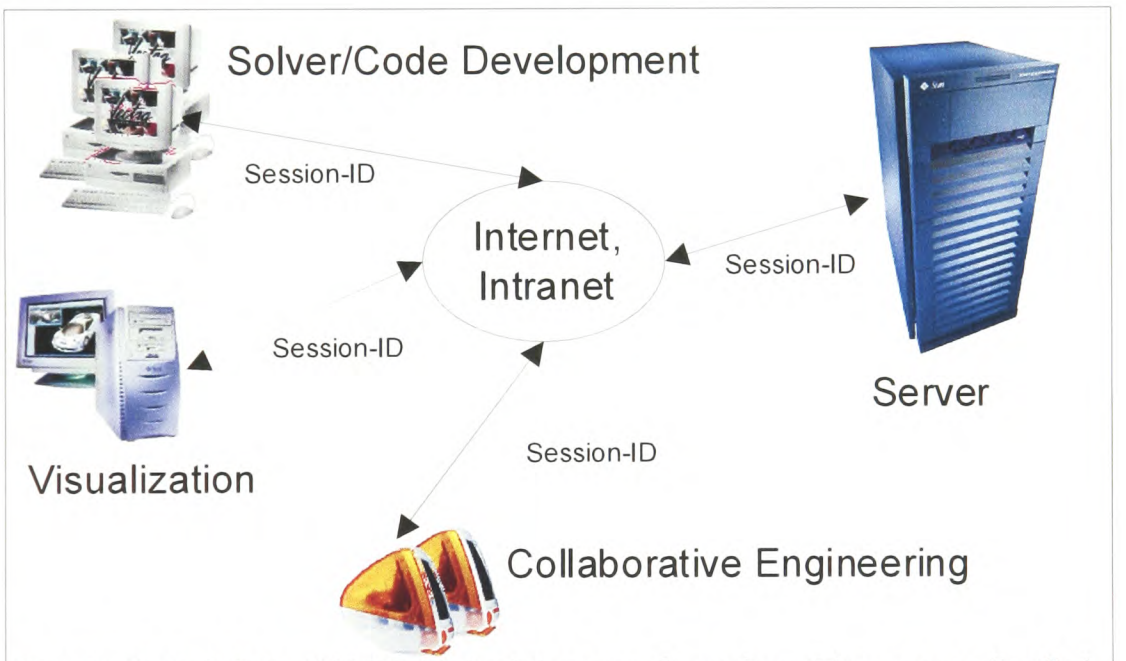


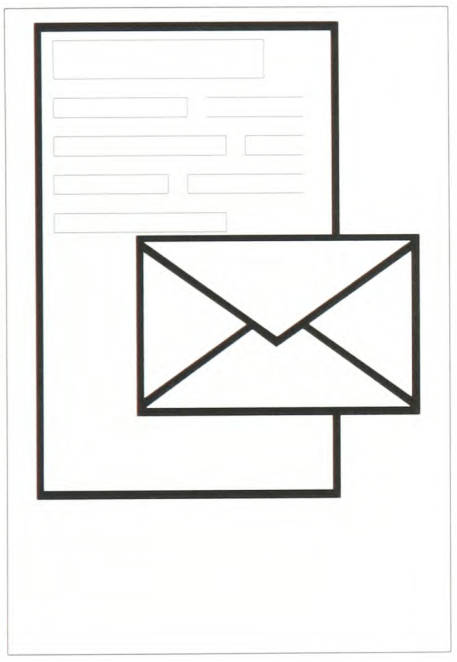
Illustration 3.3.1: *JUSTGrid Architecture Overview. The server itself, in principle, can be distributed over the internet.*

With a distributed computing system, for example an engineer at a workstation running a simulation on a supercomputer, the engineer would like to see the computation just as if it were happening on the workstation. The Java Remote Method Invocation (RMI) is one way to do this: the engineer (client) manipulates objects with a user interface, but the actions he performs (the *method invocation*) are actually performed on objects on the supercomputer (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.

If the client establishes the first connection to the server and upon requesting a new session, a random 64Bit Session-ID is created on the server and sent back to the client. Every further action to the session is bound to that Session-ID. With a valid Session-ID many clients are able to connect to the same session and give engineers the possibility to steer or visualize the

computation from many different clients (collaborative engineering). Another advantage of **JUSTGrid** is in case the internet connection to the server breaks down a client can easily reestablish the connection to the server as soon as the internet connection is up again using the Session-ID. While the internet connection is down the computation does not stop and no data will be lost. Additionally to the communication with RMI, **JUSTGrid** has also implemented a streaming server for large data files because RMI is packet oriented and inefficient for large continuous data sets.

3.4 Communication and Computation Procedure



3.4.1 Generic --static numeric-- Solver

A generic (written in C or Fortran) flow solver provides a precompiled set of functionality to an engineer. It has a static predefined functionality. All provided numerical strategies have do be declared at build time of the solver executable. The only way to extend the system with new functions is to compile and link the changed source code again. If you do not have the source code of the solver (e.g. a commercial flow solver) one cannot extend the system.

The user data will be handled like filling out a predefined form.

Illustration 3.4.2: Dynamic JUSTGrid Solver sending numerics

3.4.2 Dynamic JUSTGrid Solver

The JUSTGrid framework can also provide, like a generic solver system, predefined functionality to the engineers. But in addition, it provides the availability to send user specified numerics (solver) to the framework. It is possible to change the numeric code during the runtime of a computation at multiple times.

3.4.2.1 Sending the numerics

The JUSTGrid standard way is first sending the numeric to the JUSTGrid server.

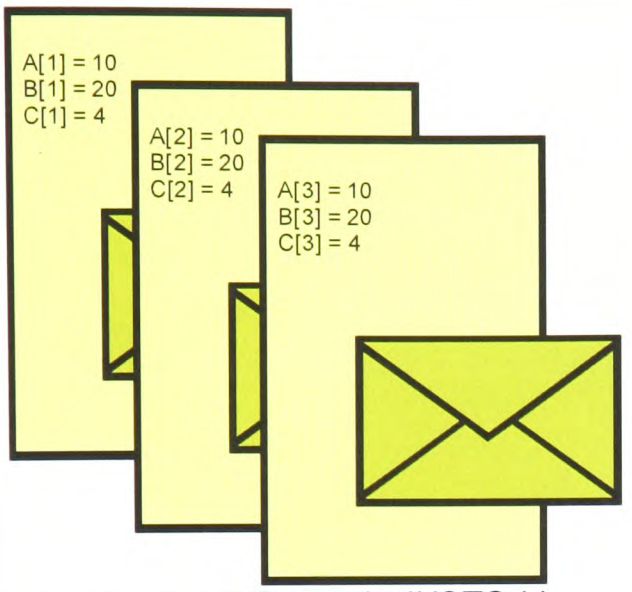


Illustration 3.4.3: Dynamic JUSTGrid Solver sending data

3.4.2.2 Sending the data

The next step is sending all data needed by **your** solver for the computation. The JUSTGrid Framework can receive additional data any time the solver requests them.

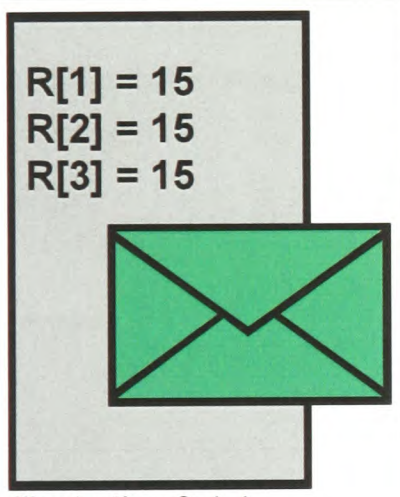


Illustration 3.4.4: Dynamic JUSTGrid solver receiving your self-defined result

3.4.2.3 Receiving the result

Receiving the results: The last step is receiving **your self defined** results. As a matter of of fact, of course the JUSTGrid framework is able to send back as many results and at any iteration one want.

3.5 Session API

The following Illustrations for the Session, Server, ... APIs are created using UML.

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

3 Multiphysics Framework - JUSTGrid

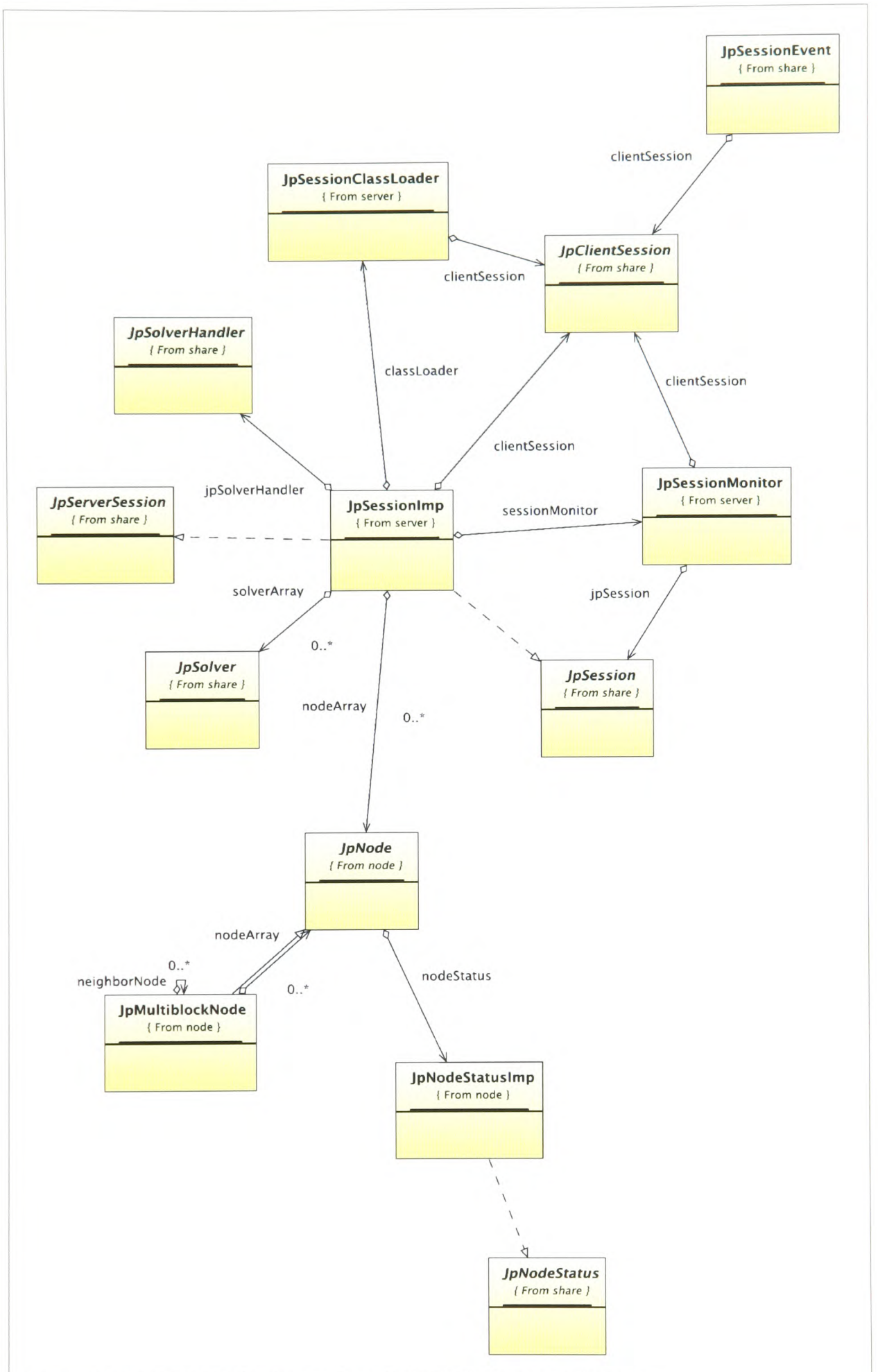


Illustration 3.5.1: UML digram for JUSTGrid Session classes

The Session API is responsible for Java class loading and transmitting and receiving all information needed by the server to run a compute session. It deals also with the event and solver handlers for interactive steering.

- *JpSessionImp* is the implementation of the JpSession and the JpServerSession interfaces on the server side. It is the central object for one complete simulation. It interacts with the client over the network, initializes all solvers and handles the complete IO.

- *JpClientSession* is the counter part of JpSession on the client side. It is responsible for the interactive steering, IO and handles the "callback" events. (e.g. the computation has finished)

- *JpClassLoader* loads all Java-Class file binaries sent from the client (e.g. JpCell, JpSolver. JpSolverHandler) into the server memory. It is also responsible for the class security.

- *JpSessionMonitor* provides online information about the state of the overall computation and also of the state of a single node.

- *JpSolverHandler* initializes the solver parameter provided by the client. It could also be used to implement additional data input or output formats.

- *JpSolver* contains the numerical implementation for one block.

- *JpNode, JpMultiblockNode* is the execution container for JpSolver. It initializes and runs the computation, does the boundary exchange, and finalizes the computation.

- JpNodeStatus, JpNodeStatusImp gives information about the current state of a node and implements the synchronization with the attached neighbours.

3.5.1 Solver

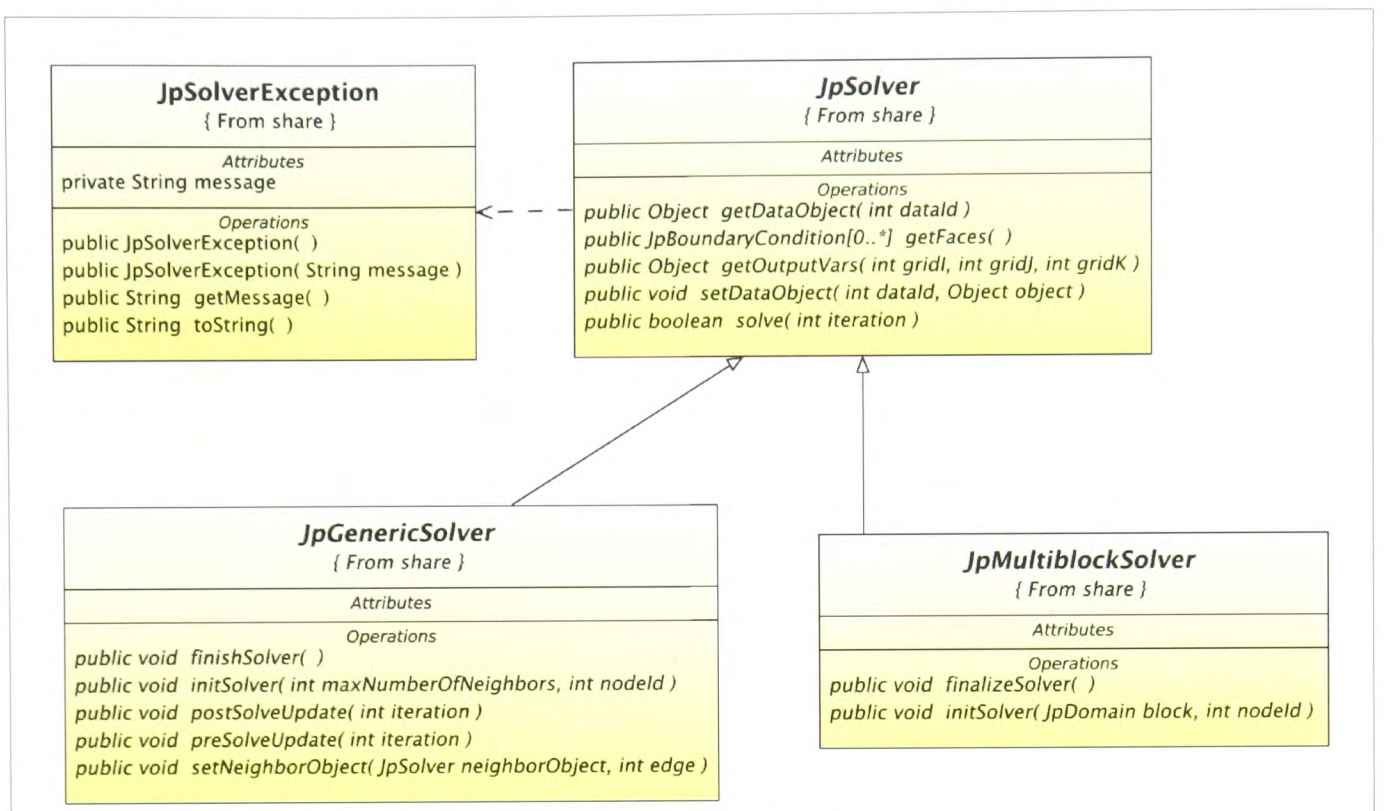


Illustration 3.5.2: UML Diagram for the JUSTGrid Solver Interface

The solver api provides the interfaces for different types of solvers. For example since the *JpGenericSolver* has no connection to any structured grid data, the user himself must implement all data communication (e.g. this is the case for the Mandelbrot-Set solver). In general, solvers implement the *JpMultiblockSolver* interface and thus get all the advantages of the ***JUSTGrid*** framework. Both solver types can throw *JpSolverExceptions* exceptions if an error occurs while running the computation.

3 Multiphysics Framework - JUSTGrid

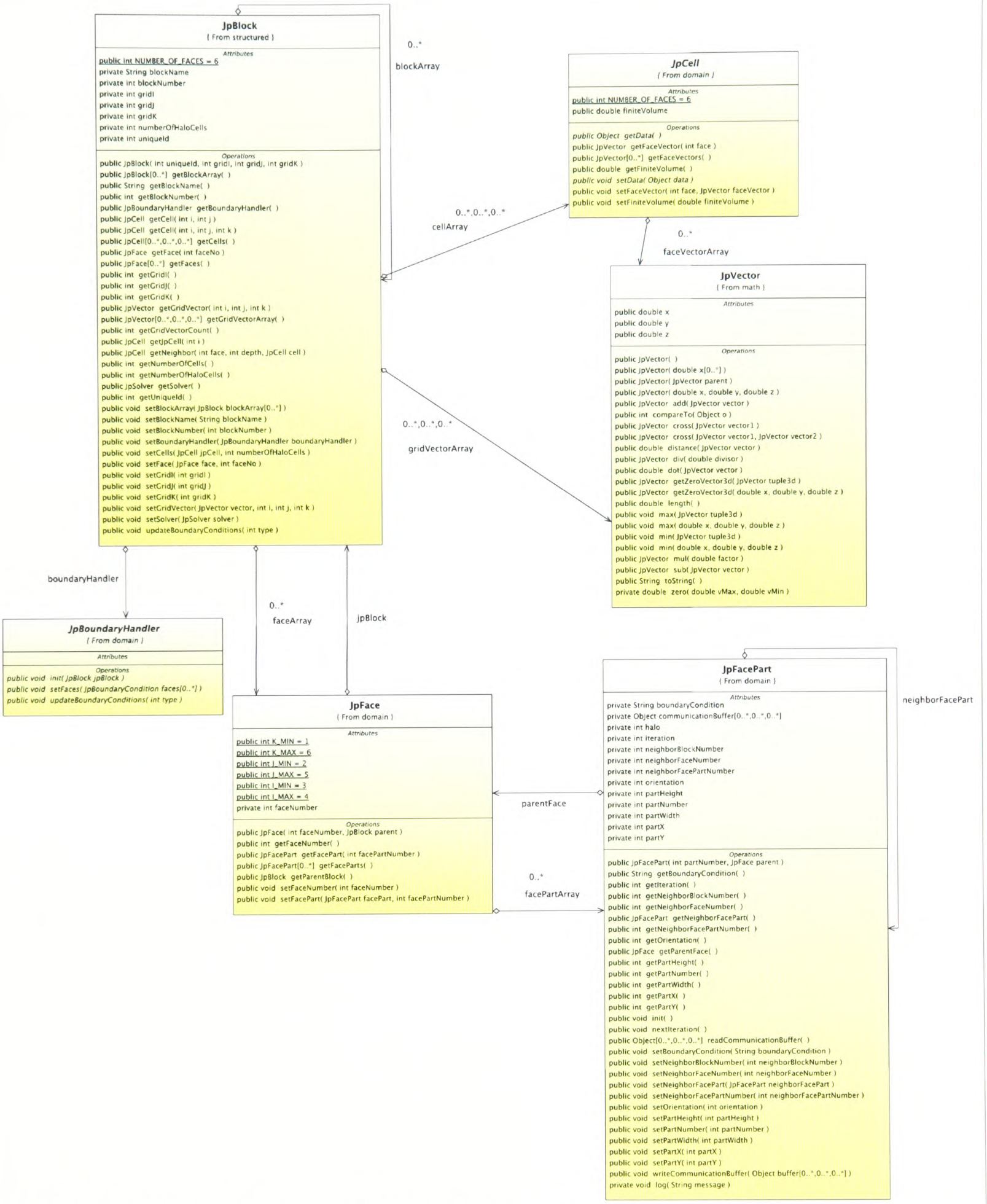


Illustration 3.5.3: UML class diagram of the JUSTGird multiblock implementation

Illustration 3.5.1 shows the complete internal class structure representing a structured grid. For detailed information and illustrations see chapter 4.3.2 on page 76.

3.5.2 Cell

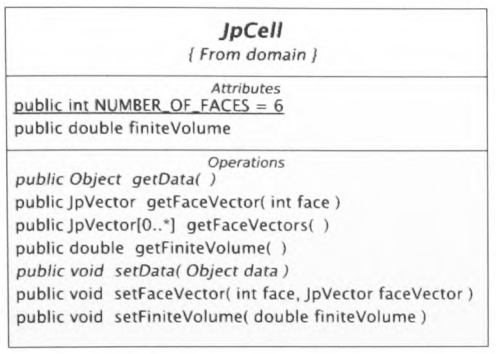


Illustration 3.5.4: The JpCell class represents one cell in a solution domain.

JUSTGRID initializes all JpCell objects for every block. JpCell contains no geometrical information but the normal vector for every cell face and the finite volume for this cell. Each face normal vector points in the normal direction and the vector length represents the face area.

3.5.3 Boundary Handler

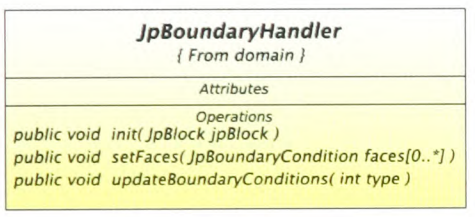


Illustration 3.5.5: This interface must be filled out for the different boundary conditions.

A boundary handler is associated to each block. The boundary handler will be executed by the ***JUSTGRID*** framework before every single iteration of the solver main compute method.

3.5.4 Session

Illustration 3.5.6: The Session object is the interactive steering interface between the client application and the server

The *JpSession* interface is the central object between the sever and the client. *JpSession* is implemented as an unicast remote object which allows to remotely communicate with this object via Java Remote Method Invocation (RMI) over the internet. It is responsible for initializing the computation, data exchange and interactive steering.

3.6 Standalone Server (JpMaster)

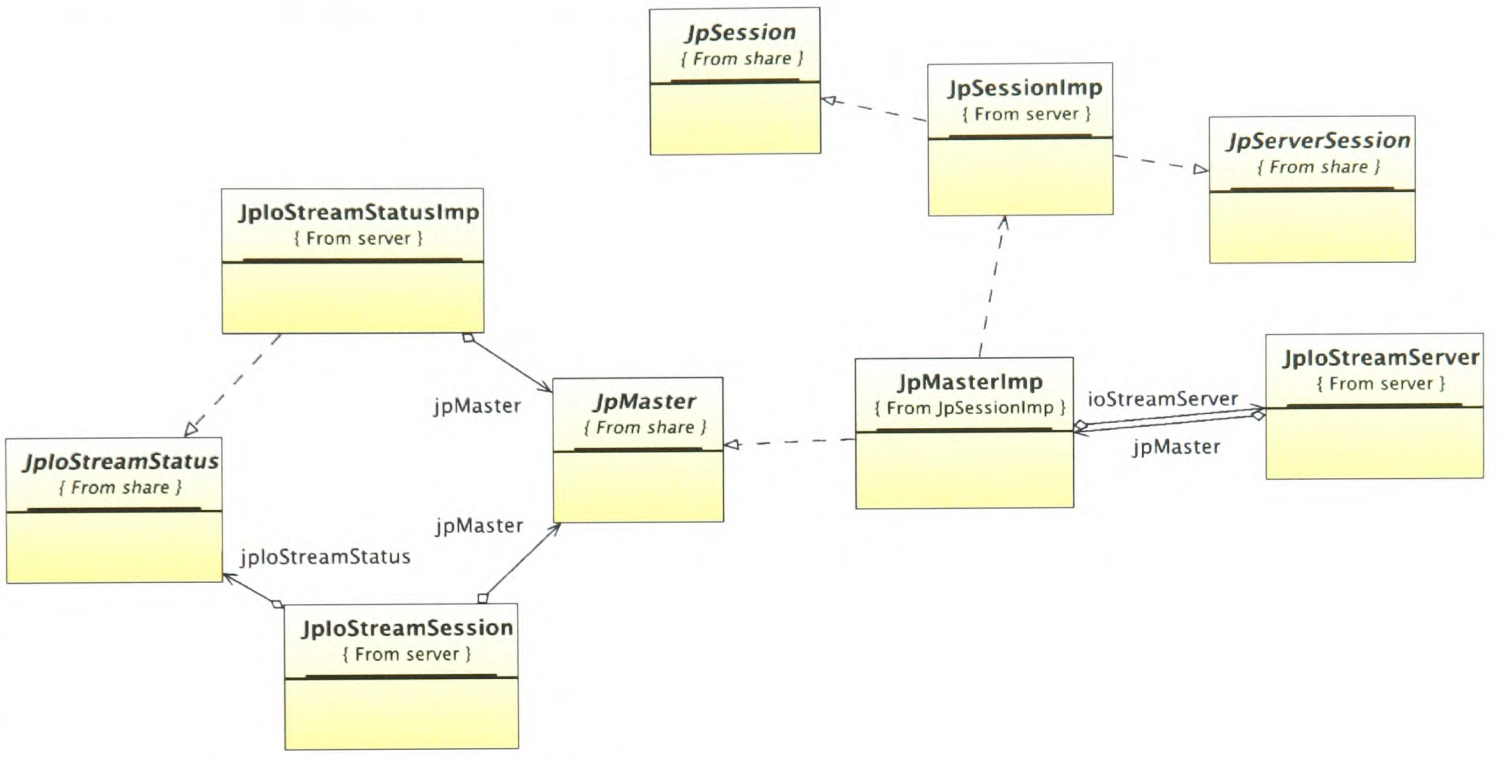


Illustration 3.6.1: UML diagram of the JUSTGrid Server classes

JpMaster is the standalone server programme running typically on a large compute system with (hopefully) a large number of processors. The JpMaster is able to exchange the default solver with a user supplied solver during a computation. JpMaster handles different solvers in different session simultaneously.

3.7 Client Applications

3.7.1 Command Line Interface

The command line interface is provided by the `just-fw.jar` file. This file contains all JUSTGrid classes. The interface is invoked as follows:

```
java hpcc.just.app.cli.Main
```

The command line interface reads a file called `startup.properties` which contains all information needed by JUSTGrid to run a compute session. For detailed information about the `startup.properties` file see chapter 4.4 on page 78.

3.7.2 Simple Client

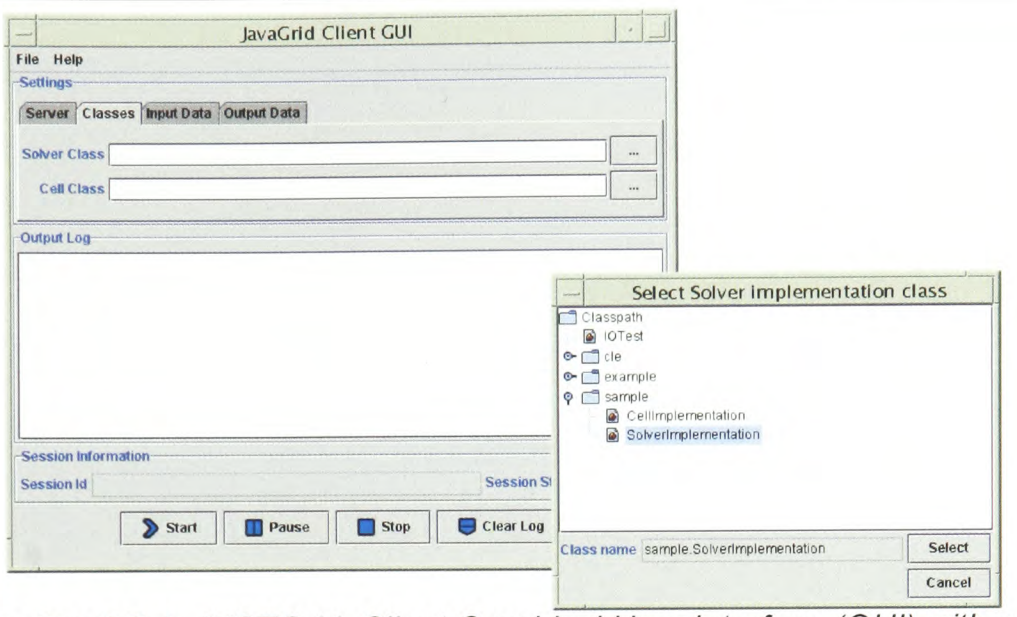


Illustration 3.7.1: JUSTGrid: Client Graphical User Interface (GUI) with an opened class browser dialog for selecting the solver class to be used

This GUI provides the interface between the user and the *JUSTGrid* collecting all information necessary to run the parallel application. In addition, 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.

3.7.3 ShowMe 3D

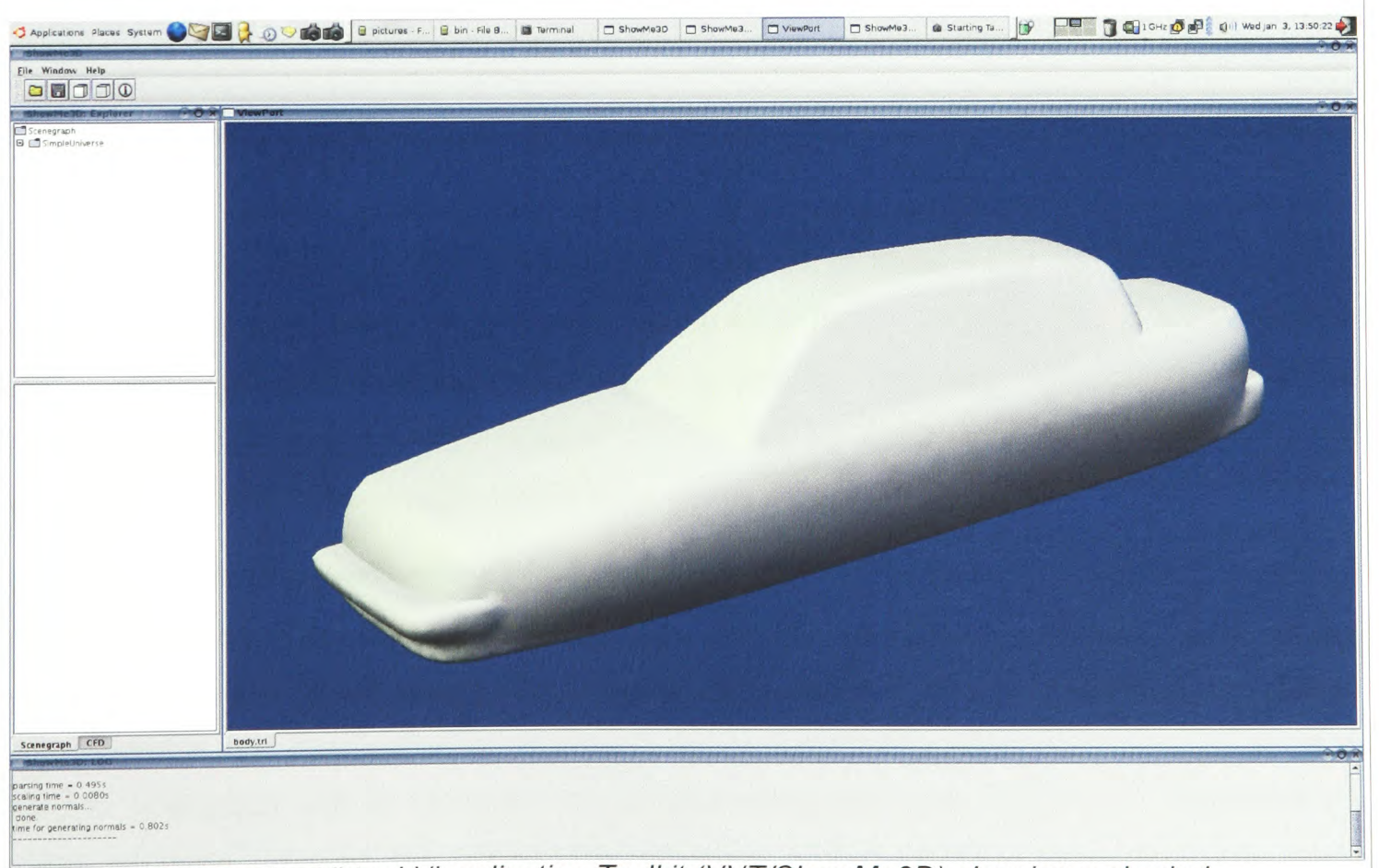


Illustration 3.7.2: The Virtual Visualization Toolkit (VVT/ShowMe3D) showing a shaded triangulated surface of a generic car.

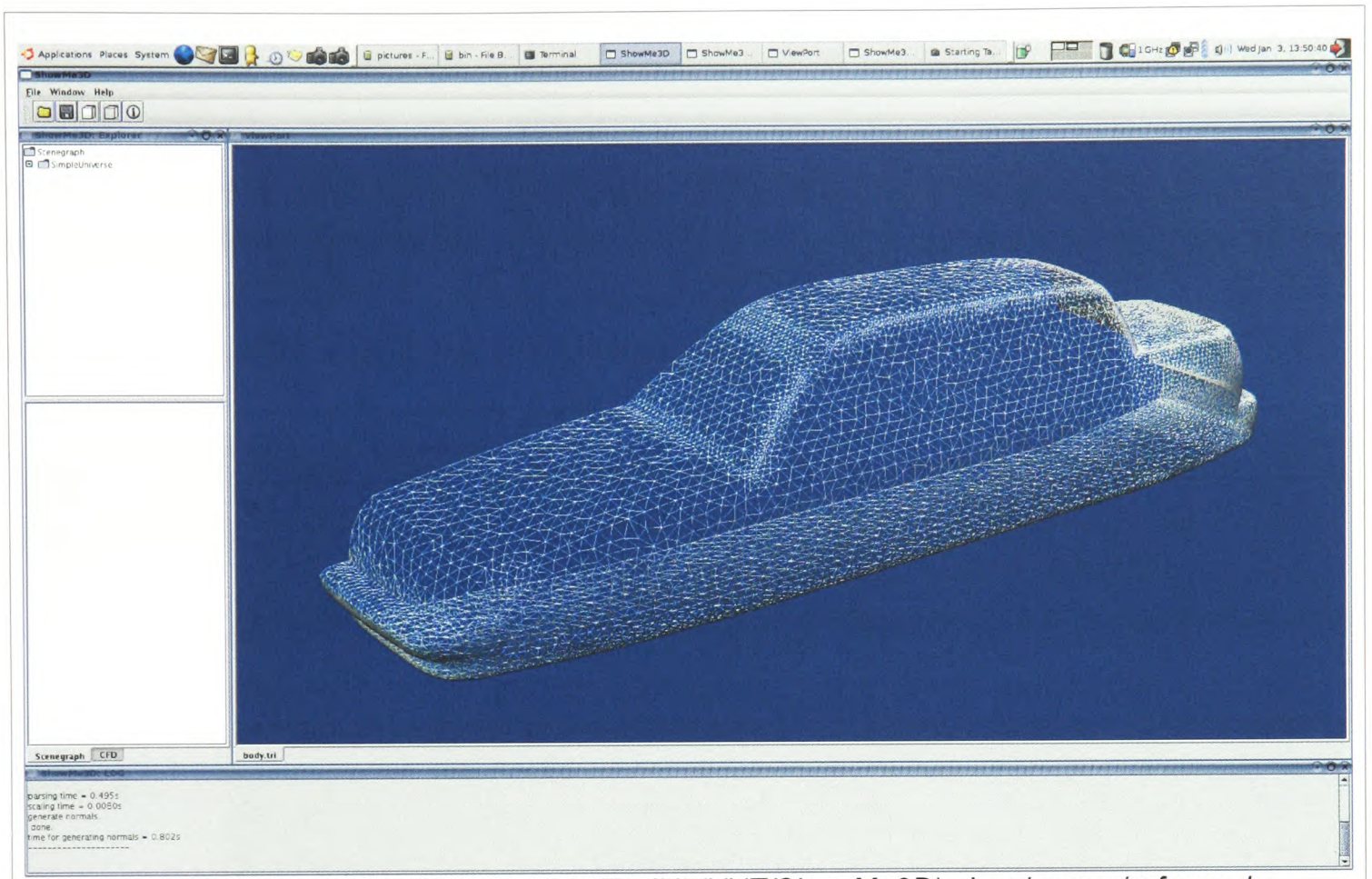


Illustration 3.7.3: The Virtual Visualization Toolkit (VVT/ShowMe3D) showing a wireframed triangulated surface of a generic car.

This program provides a way to visualize and investigate complex simulations with a thin client; that is, a machine with just a normal web browser and a low-speed connection to the internet. The client is not assumed to have expensive and complex visualization software installed. The files for the simulation data as well as the visualization software are installed on a powerful server machine.

In JUSTGrid 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 will be developed, based on the Java3D standard.

In large simulations, grids with millions of cells are computed, producing hundreds of megabytes of information during each iteration. Depending on the numerical 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 (see Illustration 3.3.1).

3 Multiphysics Framework - JUSTGrid

A suitably authenticated client sends a request that 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, which may contain file names, filtering commands, and the type of visualization software to be used. At present the widely used graphics packages Tecplot and Ensight are considered for this role. The bulk of the request is in the scripting language used by the chosen software, containing camera angles, ISO surface values, colors, and so on; that is all the information required to build one or more images of the flow.

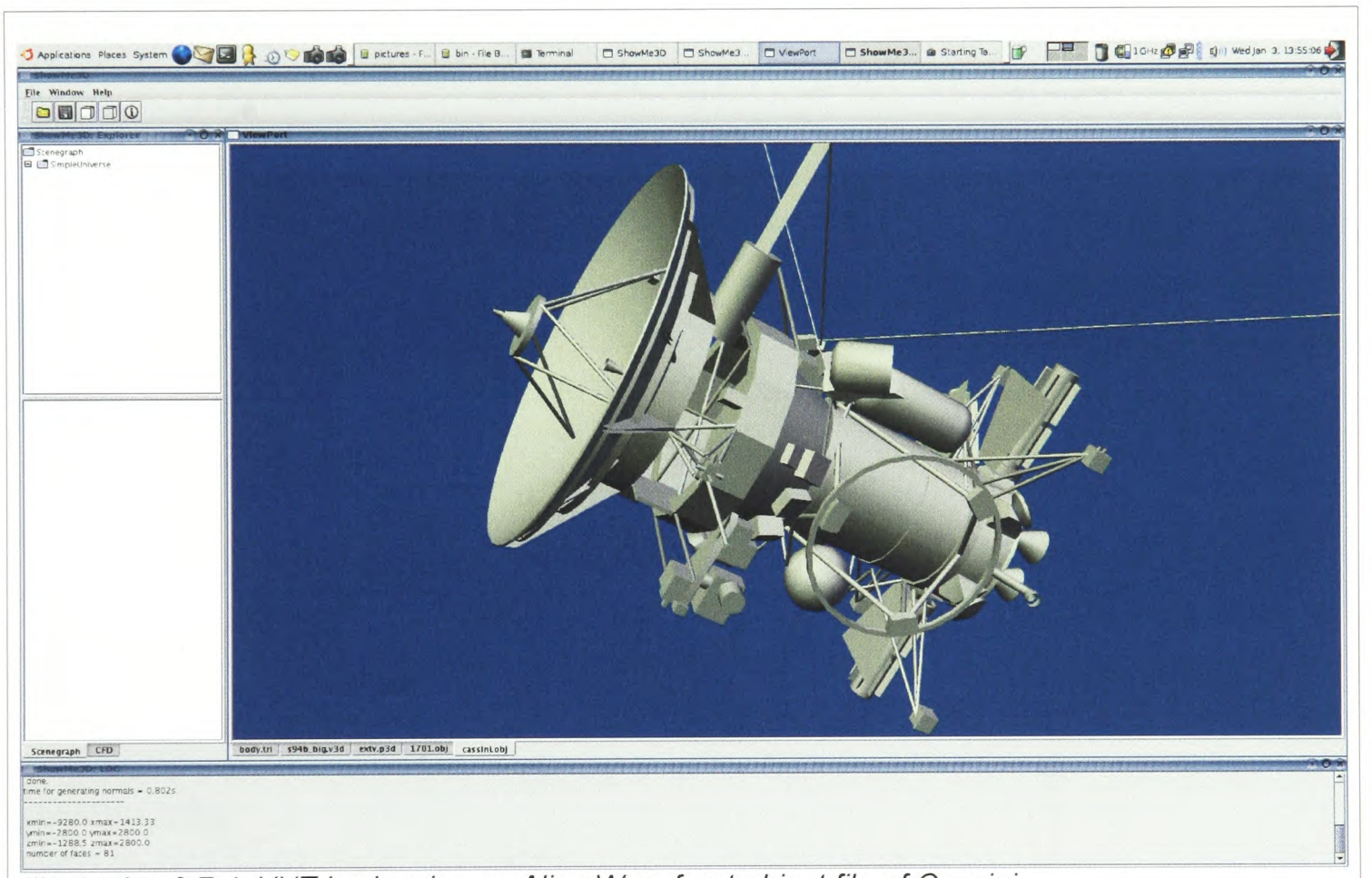


Illustration 3.7.4: VVT is showing an Alias Wavefront object file of Cassini

Clients with more powerful machines and/or a high bandwidth connection to the server might like more than images. In addition, one can 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). A client could, for example, select a density ISO surface 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.

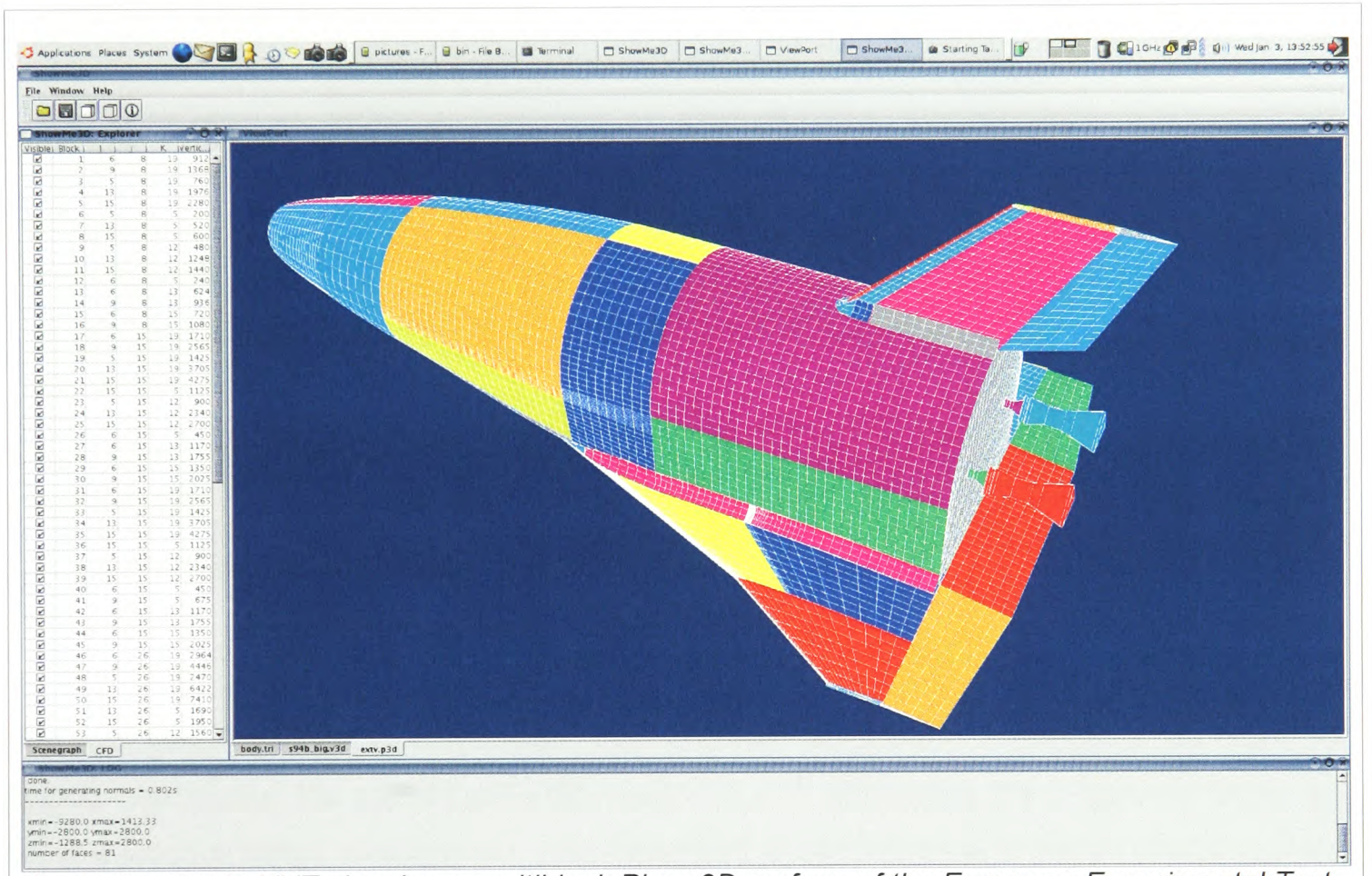


Illustration 3.7.5: VVT showing a multiblock Plane3D surface of the European Experimantal Test Vehicle (EXTV)

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.

3.7.4 GRX Monoblock Tool

3.7.4.1 GRX Monoblock Tool - 2D

This ***JUSTGrid*** simple frontend (Illustration 3.7.6) is a rapid prototype to demonstrate the simplicity of a well designed GUI for a 2D mono block Euler solver. It converts GridPro™ and TecPlot™ grid files into GRX file format. (see chapter A on page 151). This frontend acts also as a control center for the Euler solver.

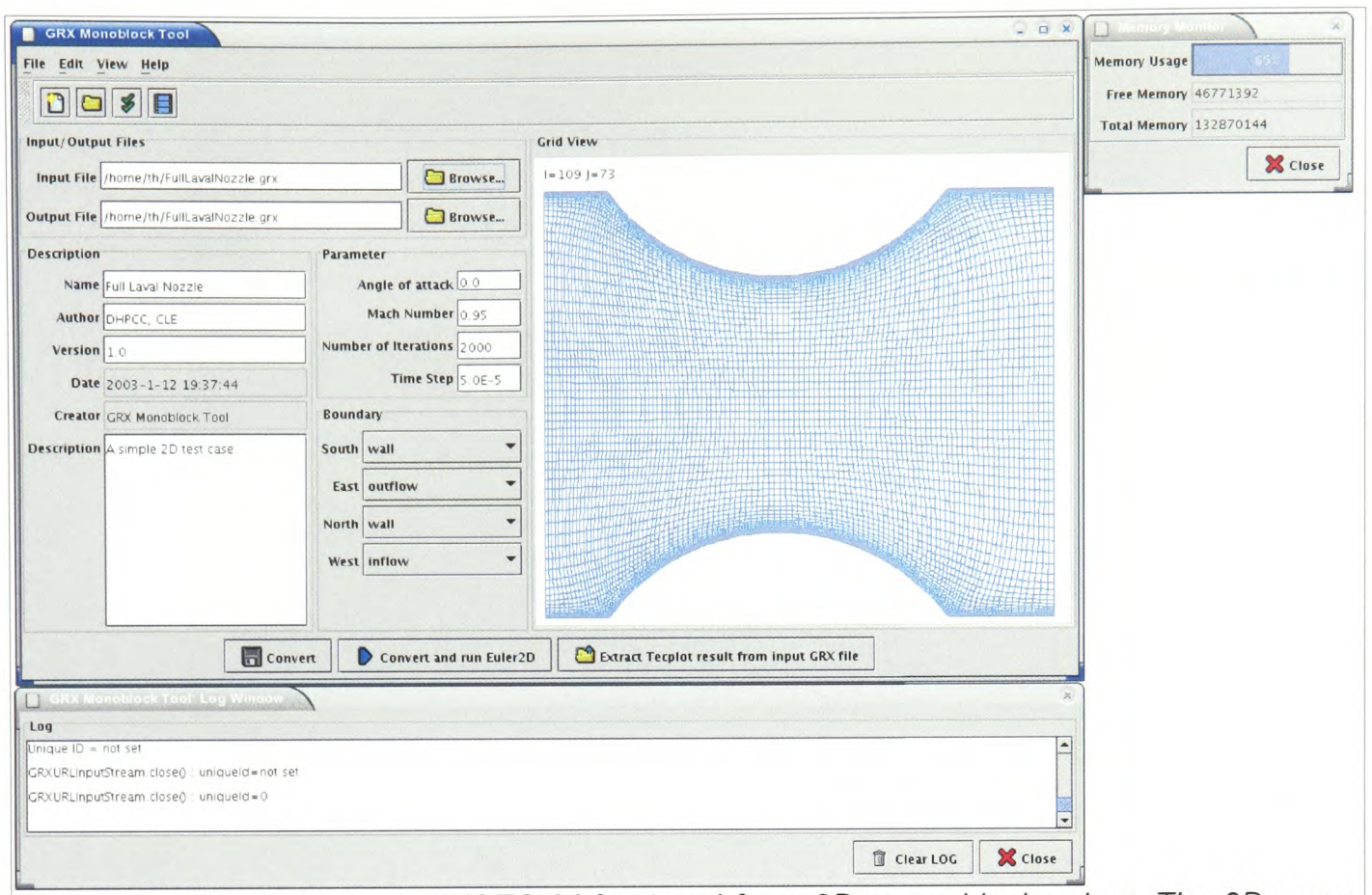


Illustration 3.7.6: *A simple **JUSTGrid** front-end for a 2D mono block solver. The 3D mono block solver is being developed. Upon testing, this solver is merged with the parallel infrastructure of the **JUSTGrid**.*

Dealing with the Java Media Framework one has the possibility to render video files from the solution domain during the computation (Illustration 3.7.7), employing the integrated video player to display the solution video in real time . The integrated video player acts like a normal video player, for instance, the usual commands, ***forward, pause and reverse playing*** are available. In ***JUSTGrid*** remote data visualization along with data compression and feature extraction as well as remote computational steering is of prime importance. Since ***JUSTGrid*** allows multiple sessions, multiuser collaboration is needed. Different visualization modules are needed, but here a computational fluid dynamics (CFD) module, allowings the perusal of remote CFD data sets is being developed, based on the Java3D standard.

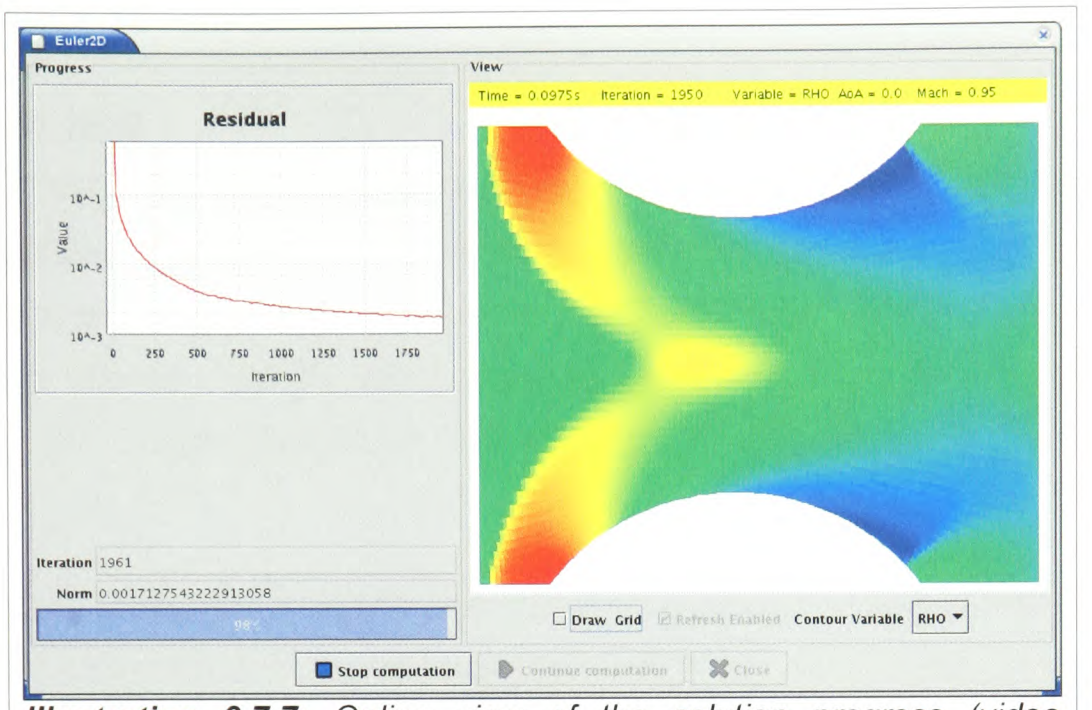


Illustration 3.7.7: *Online view of the solution progress (video production).*

3.7.4.2 GRX Monoblock Tool - 3D

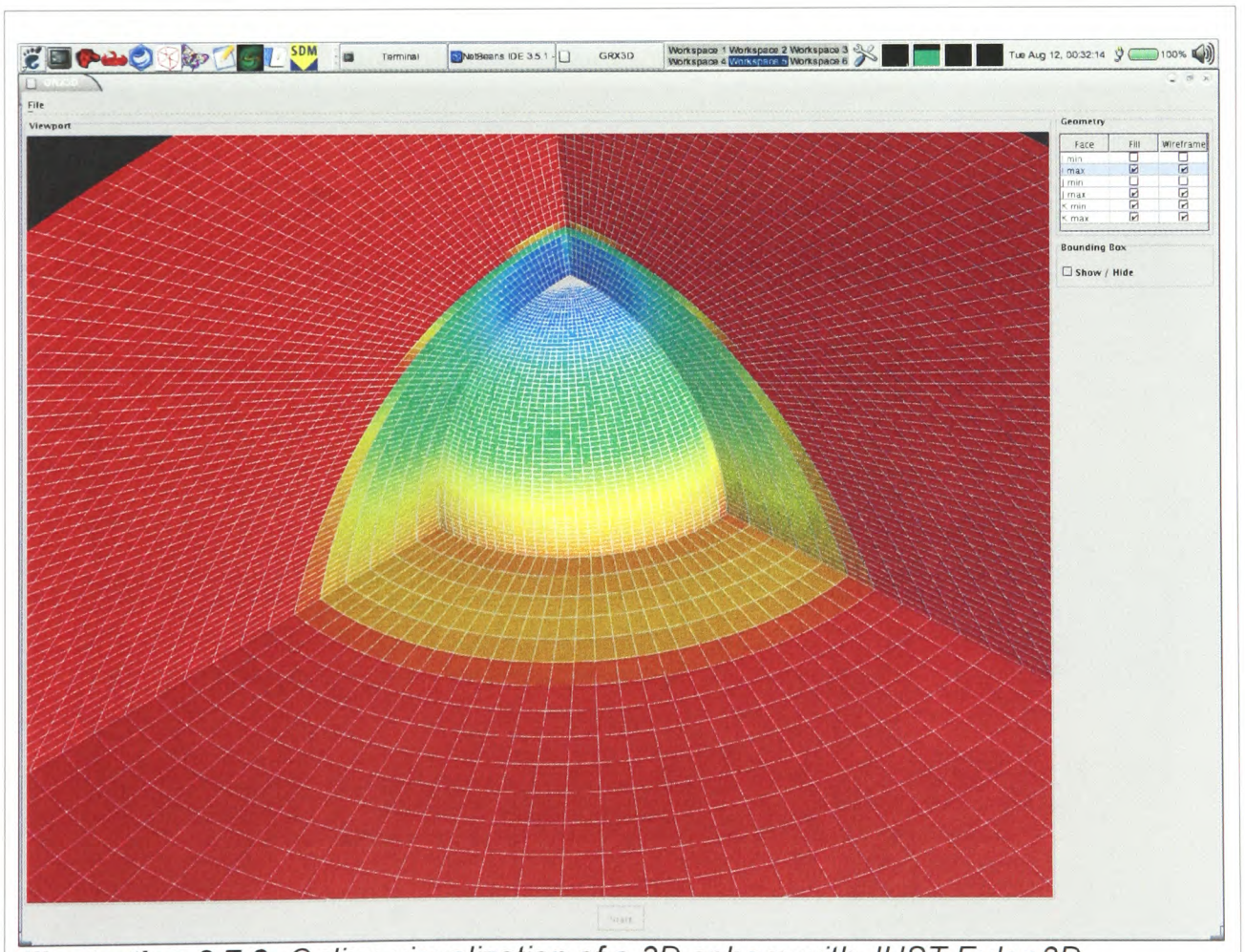


Illustration 3.7.8: *Online visualization of a 3D sphere with JUST Euler 3D*

This ***JUSTGrid*** simple frontend is a rapid prototype to demonstrate a GUI for a 3D mono block Euler solver with online visualization of the solution on specified block faces.

3.7.5 GRX Tool (multiblock)

3.7.5.1 GRX 2D Tool

The ***JUSTGRID*** GRX2D Tool can be used to prepare a simulation run. ***JUSTGRID*** GRX2D is also based on the ***JUSTGRID*** framework and uses the same loaders and utility classes as ***JUSTSOLVER*** to visualize a grid. It is the very first test to check if ***JUSTGRID*** can handle a given grid. In addition to the visualization one can specify solver specific parameters like „max number of iterations", „Mach number" or „Dt". These parameters are **not** predefined but depend on the selected solver.

GRX2D automatically scales the hole grid into the viewing area. It is able to highlight and show the bounding box of the complete gird, the single block boundaries, block numbers and even the cells.

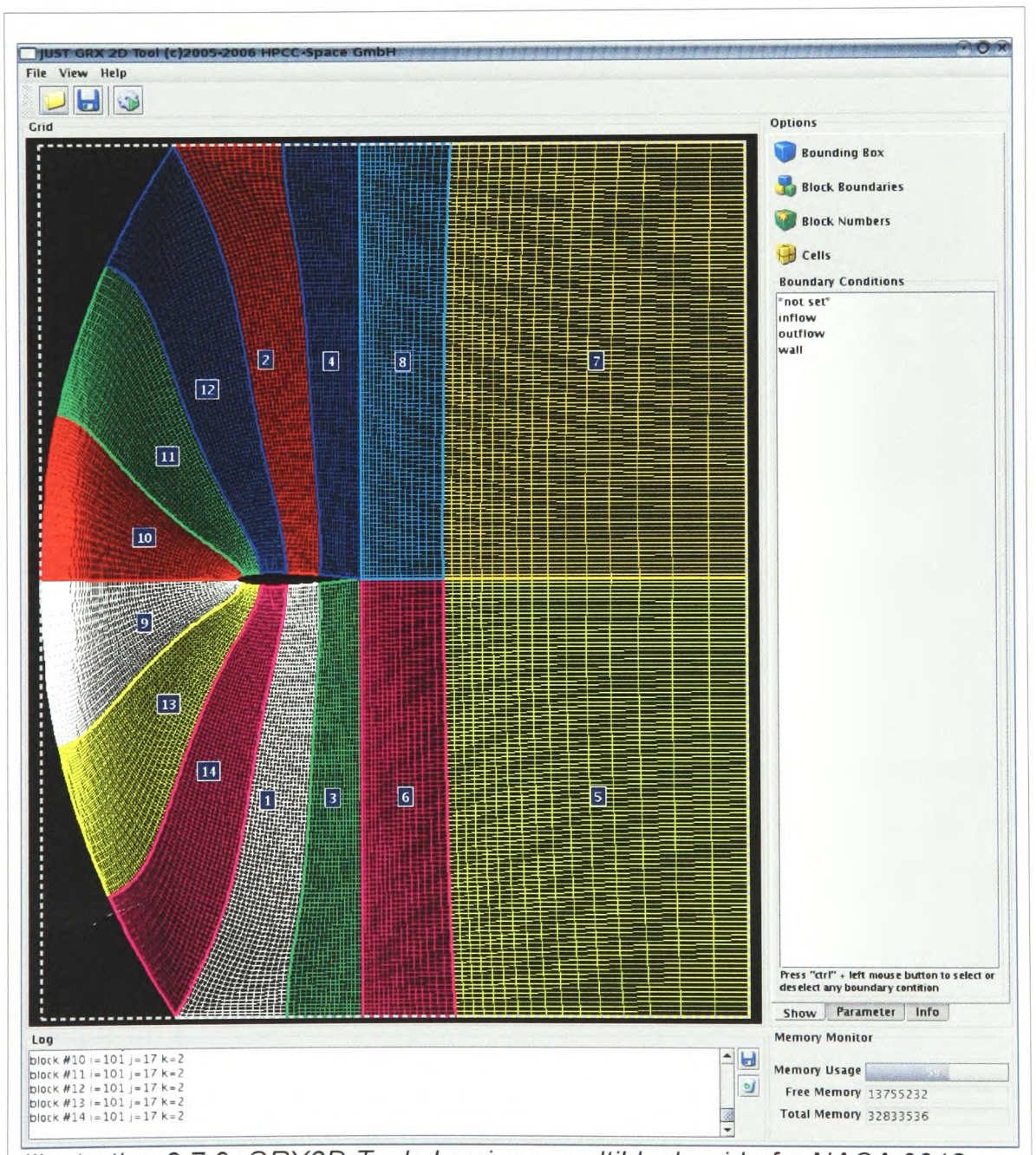


Illustration 3.7.9: GRX2D Tool showing a multiblock grid of a NACA 0012 airfoil

3.7.5.2 GRX 3D Tool

Like ***JUSTGRID*** GRX2D the ***JUSTGRID*** GRX3D Tool can also be used to prepare a simulation run. Even the 2D tool ***JUSTGRID*** GRX3D is based on the ***JUSTGRID*** framework and uses the same loaders and utility classes as ***JUSTSOLVER*** to visualize a grid. In difference to the ***JUSTGRID*** GRX2D this tool needs the Java 3D API for the visualization.

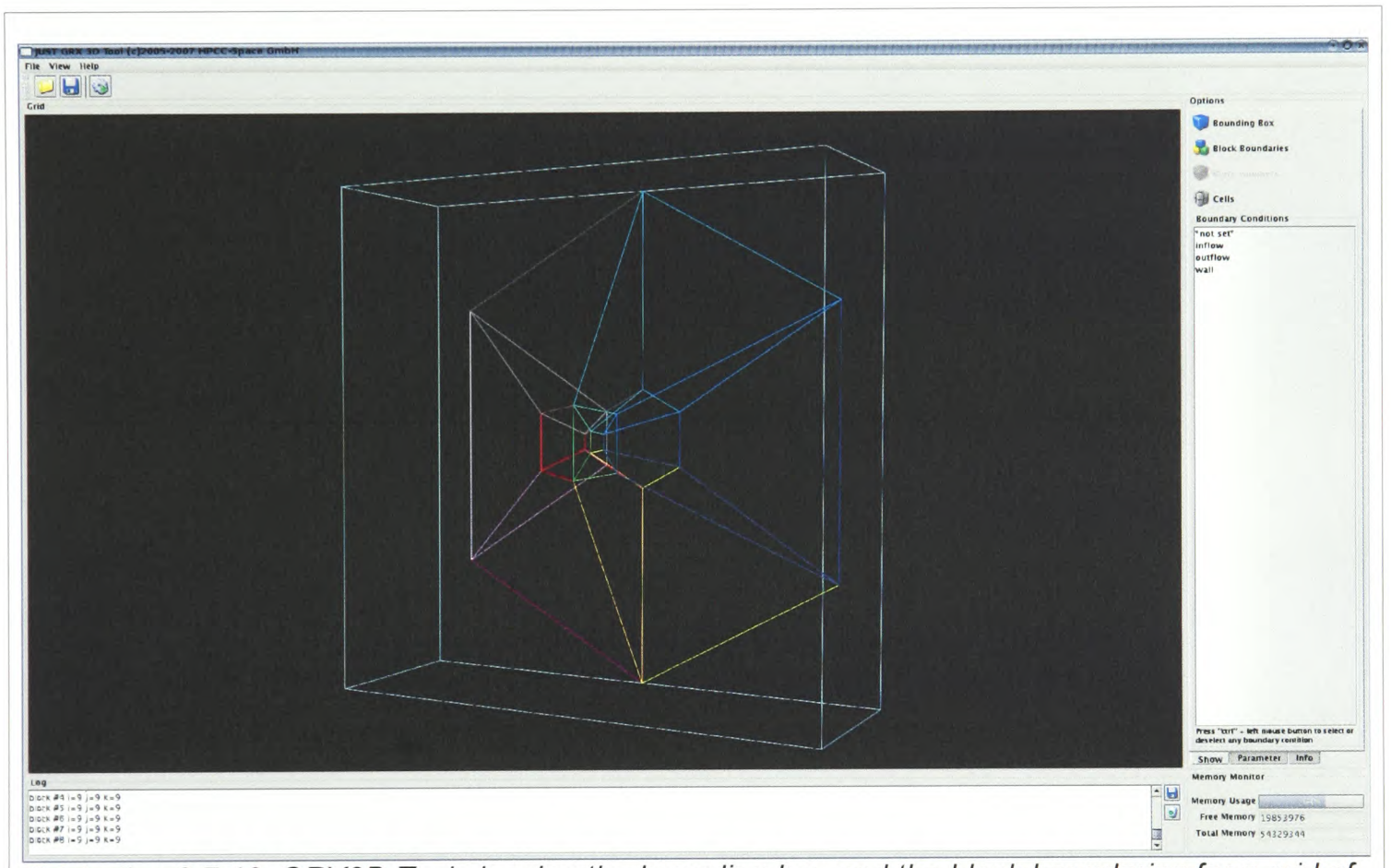


Illustration 3.7.10: GRX3D Tool showing the bounding box and the block boundaries for a grid of a sharp cone

The view can be freely moved, rotated, and zoomed by using the computer mouse. The boundary conditions specified by a boundary file or a GridPro topology/connectivity file are automatically shown at the right frame. One can select one or more boundary conditions to visualize all block faces being related to the specified boundary conditions.

Java 3D is a pure Java extension for visualizing and interacting with 3D scenes.

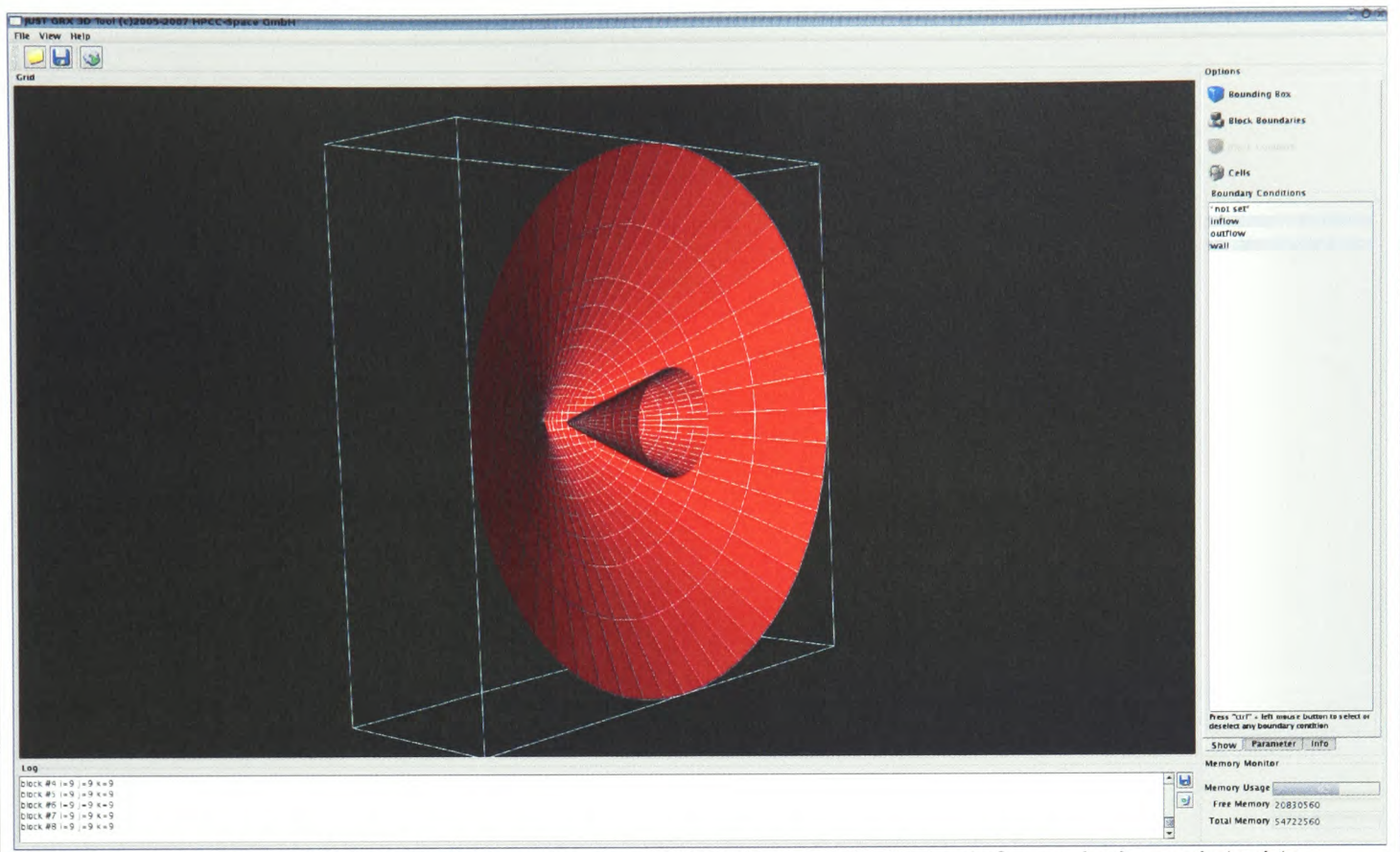


Illustration 3.7.11: GRX3D Tool showing bounding box and all block faces being related to inflow and wall boundary conditions for sharp cone grid

Java 3D is a very powerful API. These pictures are demonstrating some of its implemented capabilities.

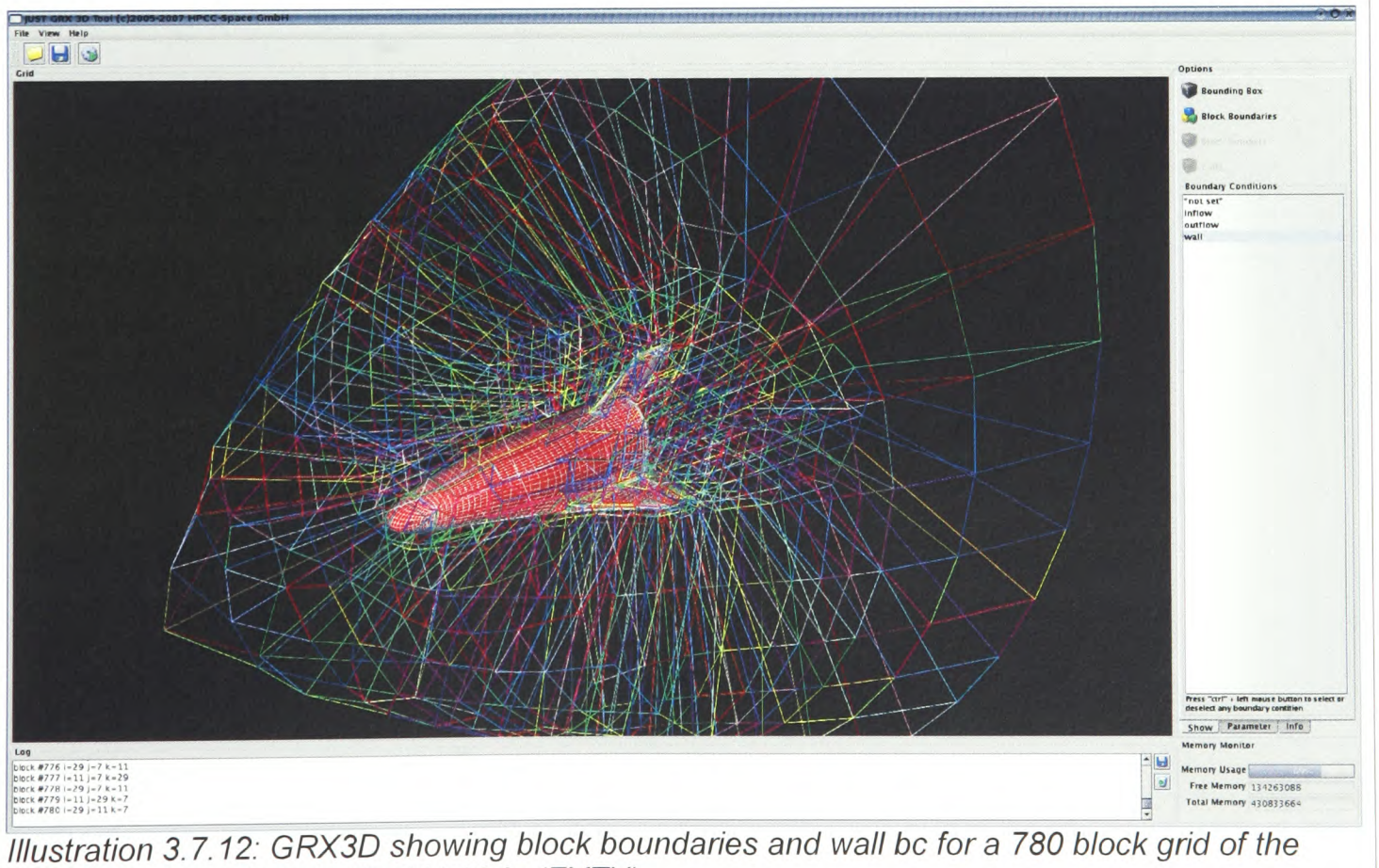


Illustration 3.7.12: GRX3D showing block boundaries and wall bc for a 780 block grid of the European Experimental Test Vehicle (EXTV)

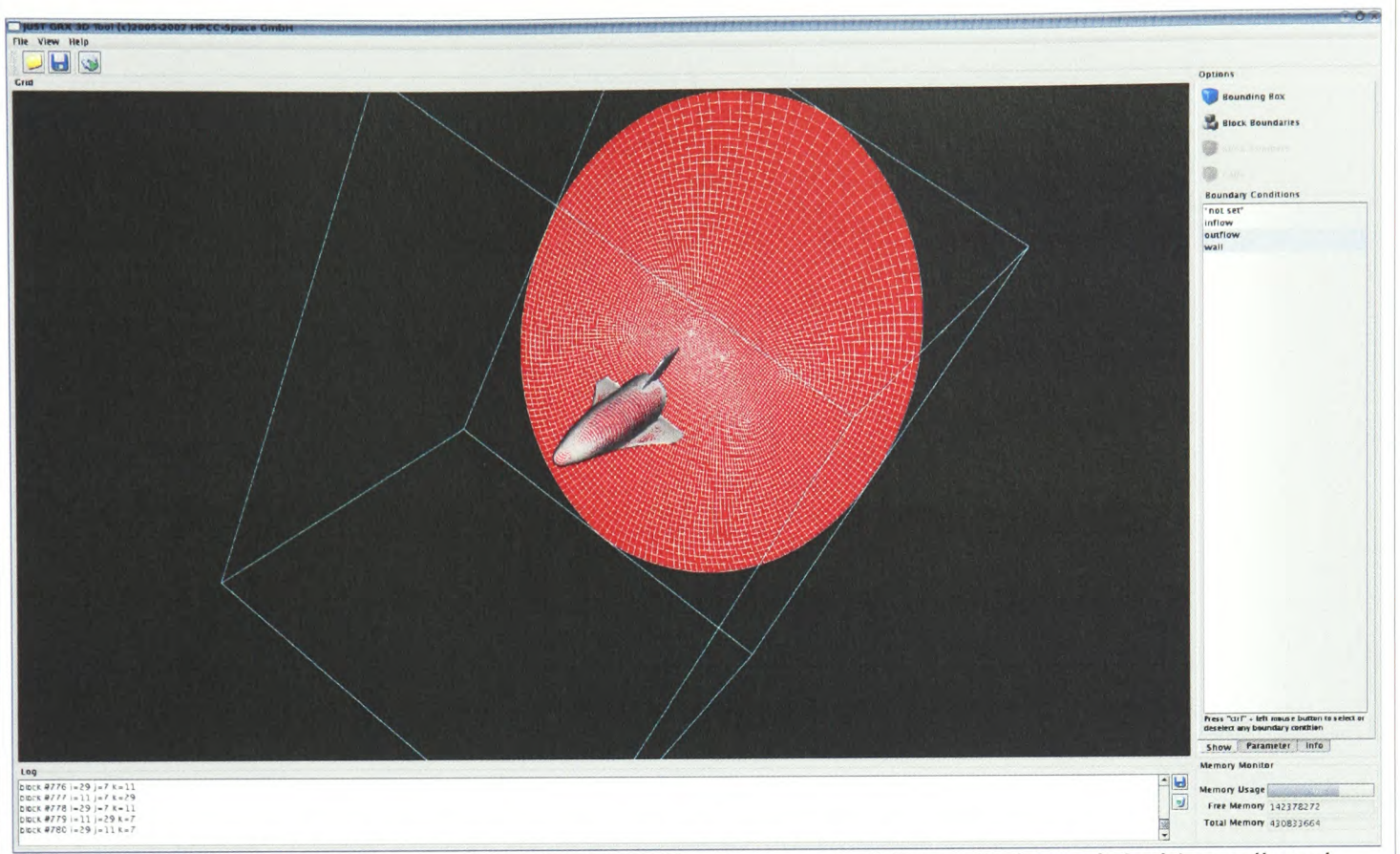


Illustration 3.7.13: GRX3D showing the bounding box and all faces being related to wall and outflow boundary conditions for a 780 block EXTV grid

It is possible to rotate, move or zoom into the loaded grid with no special hardware. A common laptop computer with a simple 3D graphics card is powerful enough to work.

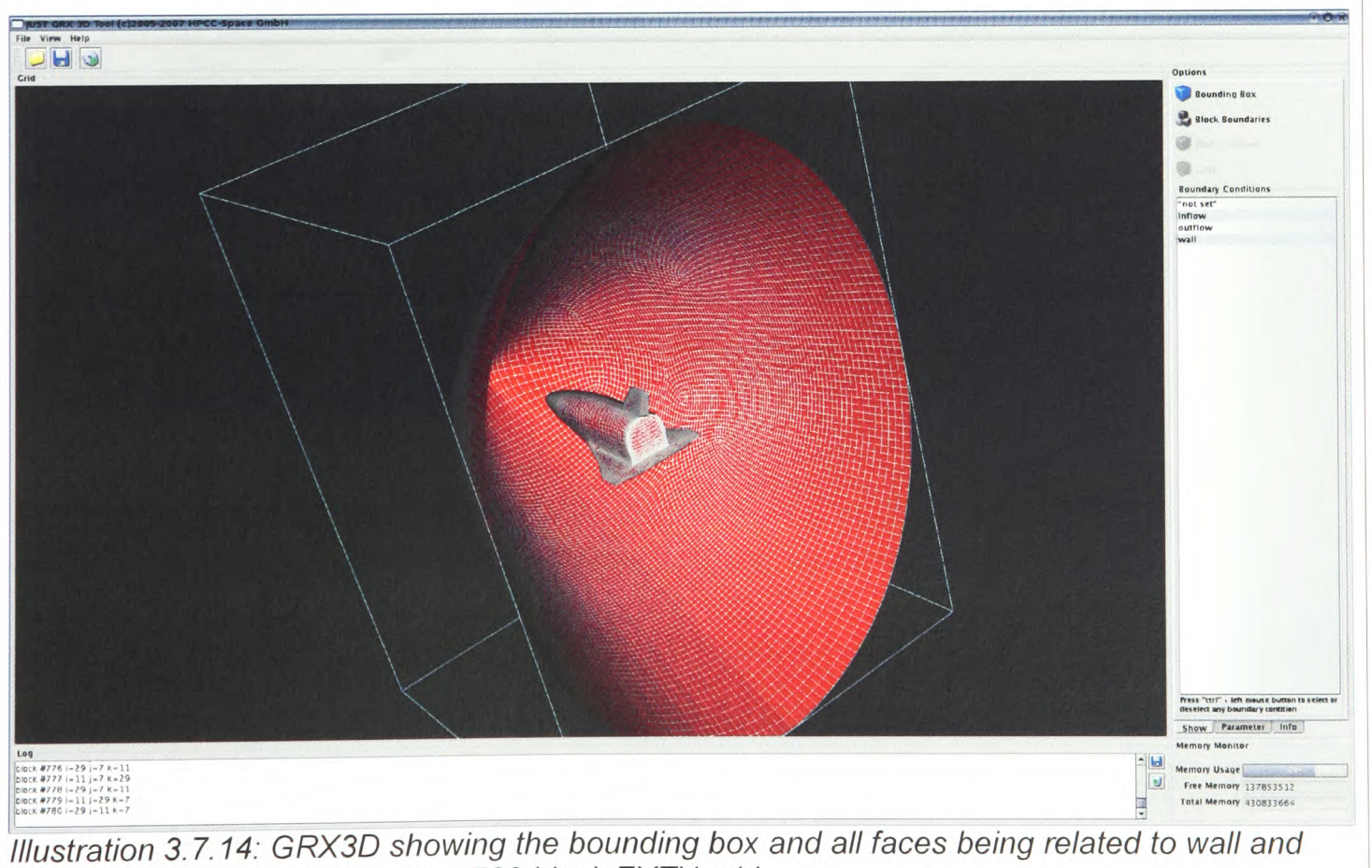


Illustration 3.7.14: GRX3D showing the bounding box and all faces being related to wall and inflow boundary conditions for a 780 block EXTV grid

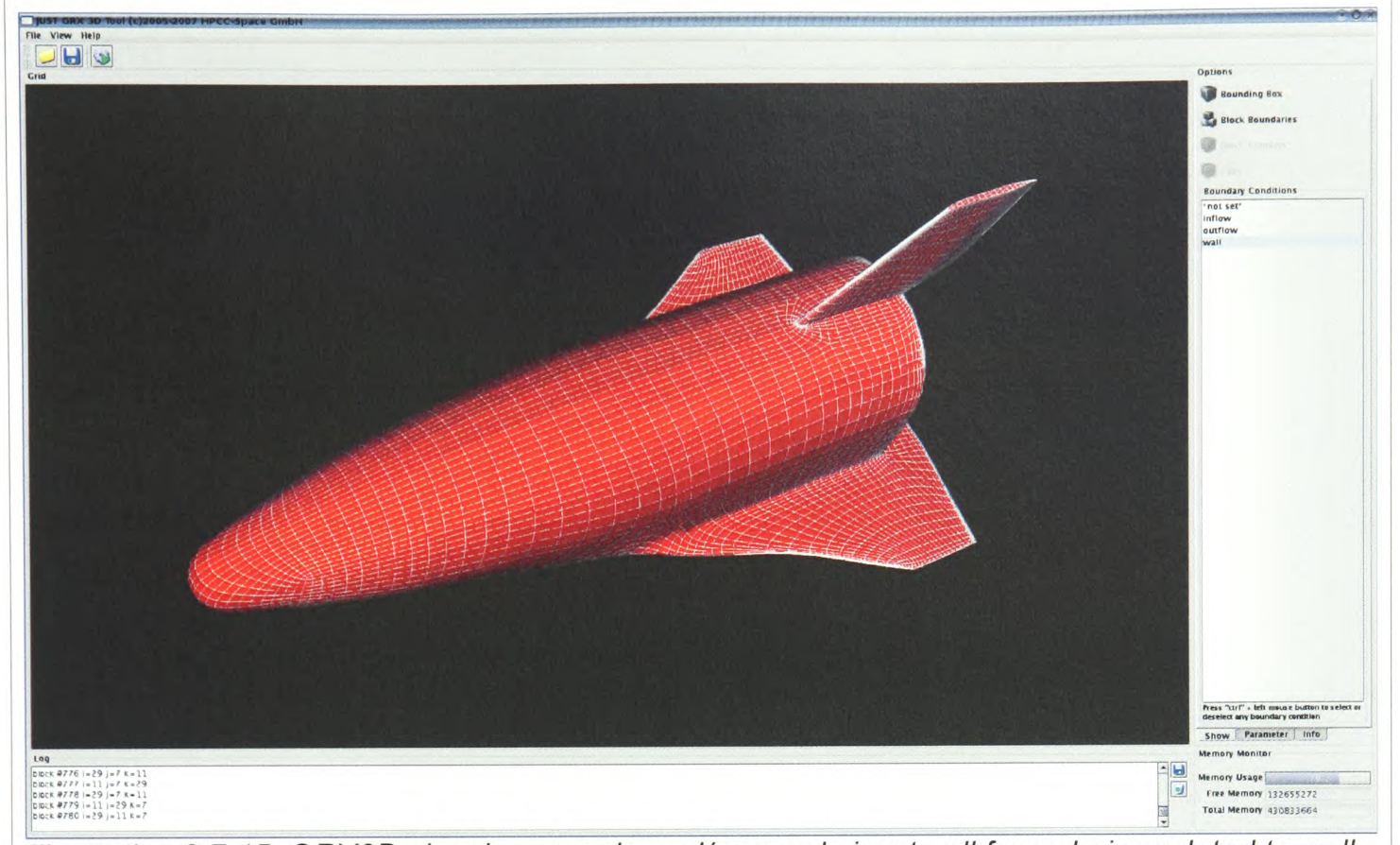


Illustration 3.7.15: GRX3D showing an enlarged/zoomed view to all faces being related to wall boundary conditions for a EXTV grid.

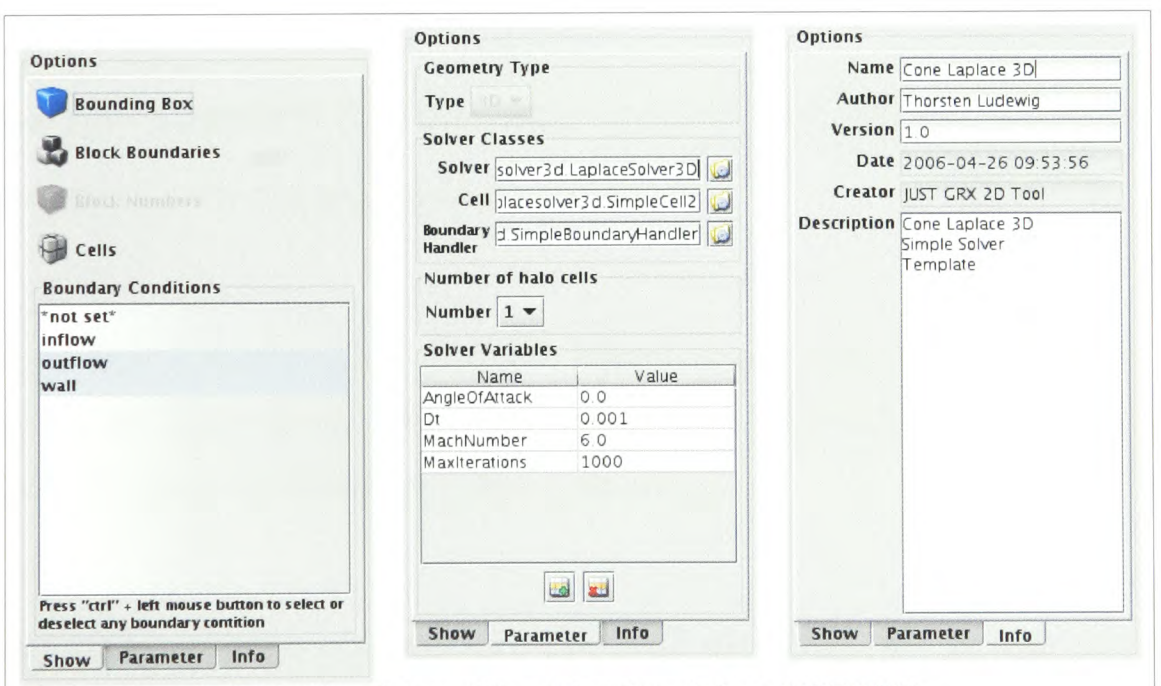


Illustration 3.7.16: The three different option tabs of GRX3D

- The “Show” option tab is responsible for the viewing area. One can specify what should be visualized.
- Within the “Parameter” tab all physical and numerical parameters needed by the simulation can be specified.
- The “Info” tab is for additional information only.

3.8 Using legacy C or Fortran Code within JUSTGrid

While one can write the solver entirely in Java, there are situations where Java alone does not meet the needs of the application. Programmers use the Java Native Interface (JNI) to write *Java native methods* to handle those situations when an application cannot be written entirely in Java. One common task is to integrate legacy C or Fortran code into the ***JUSTGrid*** framework.

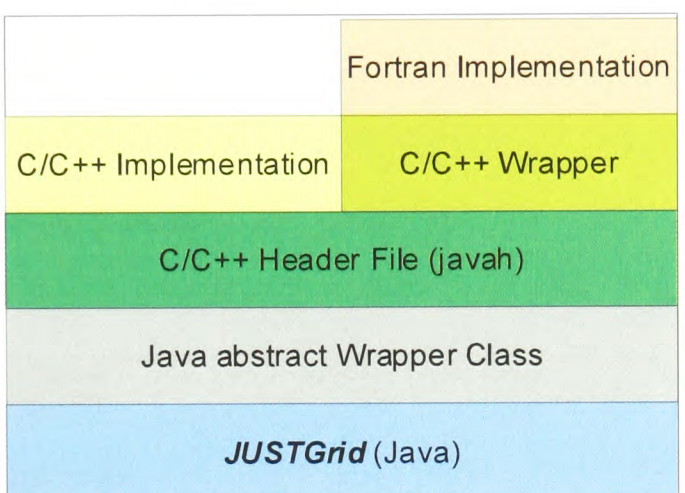


Illustration 3.8.1: Schema diagram for JUSTGrid mixing programming languages via Java Native Interface JNI

All tools to create such an environment are part of the Java Development Kit (JDK).

The difficulties of doing such integration are:

1. Data exchange between Java and the native code. Java uses a data format, which is identical for all Java Virtual Machines independent from the underlaying hardware. The data converting part could be very time consuming.

2. The native code must be provided by a dynamic link library (Windows, .DLL), shared library (Solaris, Linux, .so) or dynamic library (Mac OS X, .dylib). Static code cannot be integrated into a Java VM.

The complete integration of a sample Fortran code into ***JUSTGrid*** was tested in a prove of concept for the US Air Force. The projects working title was “witch's cauldron”.

4 Multiphysics Solver Development with *JUSTGrid*

The JUST Framework architecture is prepared for unstructured, structured and merged grids. At this time the full implementation is only available for structured grids.

4.1 Development Prerequisites

- A Java Development Kit (JDK) version 1.4.2 or higher. (http://java.sun.com)
- Java 3D API version 1.3.2 or higher. (https://java3d.dev.java.net/)
- A source editor or an Integrated Development Environment (IDE) I prefer the NetBeans IDE (http://www.netbeans.org) but you can use any editor or IDE you want.
- The ***JUSTGrid*** archive file named `just-fw.jar`
- Make yourself familiar with the following JUST Framework classes:
  - hpcc.just.domain.JpCell
  - hpcc.just.domain.JpFace
  - hpcc.just.domain.JpFacePart
  - hpcc.just.domain.structured.JpBlock
  - hpcc.just.share.JpMultiblockSolver

If your solver needs to have special initialization methods you also must have a look at the following two class definitions.

- hpcc.just.share.JpSolverHandler
- hpcc.just.client.JpGenericSolverHandler

For a better understanding of the internal classes you should also read the documentation of the mathematical vector classes.

- hpcc.math.JpPoint
- hpcc.math.JpVector
- hpcc.math.JpVectorMath

4.2 Sample integration of an Euler3D solver into JUSTGrid

1. Compile the solver as it is.
2. Run the solver with a well known example and save the result for comparison with the migrated solver.
3. Create a new NetBeans project and copy all solver classes into the source directory of this project. This is an optional task.
4. Move all solver classes into a new Java package to avoid naming conflicts.
5. Add the `just-fw.jar` archive to the projects library settings or add it to your classpath environment variable.
6. Determine or create the class files for cell, solver, and boundary handler
7. Check the order of cell array indices, they must be $[I_{min}\text{-}I_{max}]$ $[J_{min}\text{-}J_{max}]$ $[K_{min}\text{-}K_{max}]$
8. Compile the cell, solver, etc. classes
9. Create a `startup.properties` file. For more information about this file see chapter 4.4 on page 78.
10. Start

```
java hpcc.just.app.cli.Main
```

ATTENTION: Make sure that all classes of the solver are in the classpath.

11. Compare the result with the result computed in point 2 to make sure that the changes were correct.

4.3 JUSTGrid provided structure

4.3.1 Description of the Standard Cube

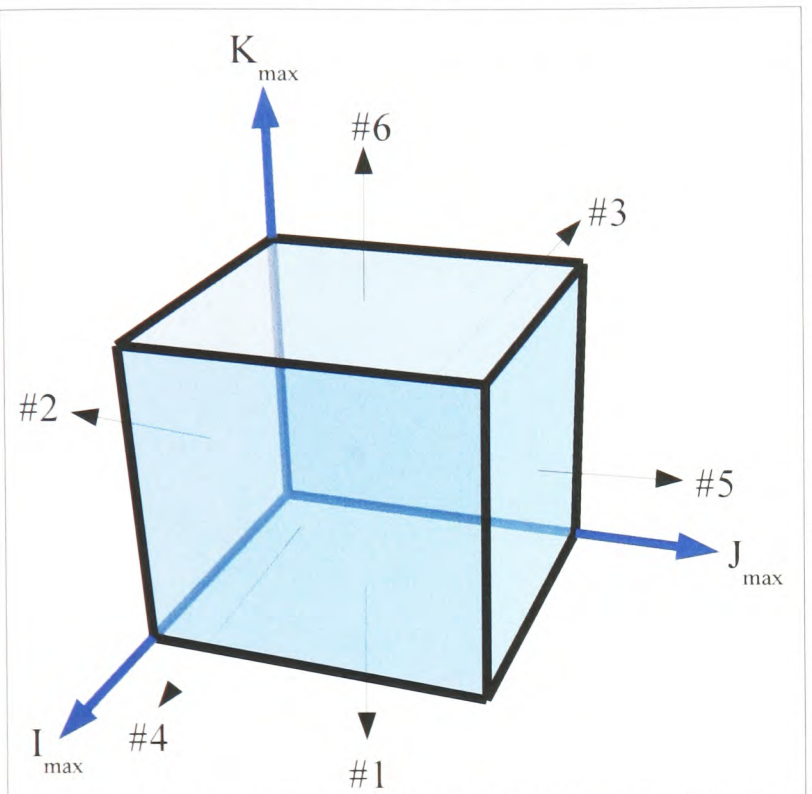


Illustration 4.3.1: Description of the Standard Cube

A formal description of block connectivity is needed to perform the block updating, i.e., to do the message passing. To this end, grid information is subdivided into topology and geometry data that are kept separate. The following format is used for both the grid generator and the flow solver, using the same topology description. All computations are done for a standard cube in the computational plane as shown in Illustration 4.3.5. The coordinate directions in the computational plane are denoted by I,J, and K and block dimensions are given by I_{max}, J_{max} and K_{max}, respectively.

In the computational space, each cube has its own right-handed coordinate system (I,J,K), where the I direction goes from back to front, the J direction from left to right, and the K direction from bottom to top, see Illustration 4.3.5. The coordinate values are by proper grid point indices i, j, k in the I, J, K directions, respectively. That means that values range from 1 to I_{max} in the I direction, from 1 to J_{max} in the J direction, and 1 to K_{max} in the K direction. Each grid point represents an integer coordinate value in the computational plane.

A simple notation of planes within a block can be defined by specifying the normal vector along with the proper coordinate value in the specified direction. For example, face 2 can uniquely defined by describing it as a J plane with a *j* value 1 i.e., by the pair (J,1) where the first value is the direction of the normal vector and the second value is the plane index. Thus, face 4 is defined by the pair (I,J). This notation is also required in the visualization module.

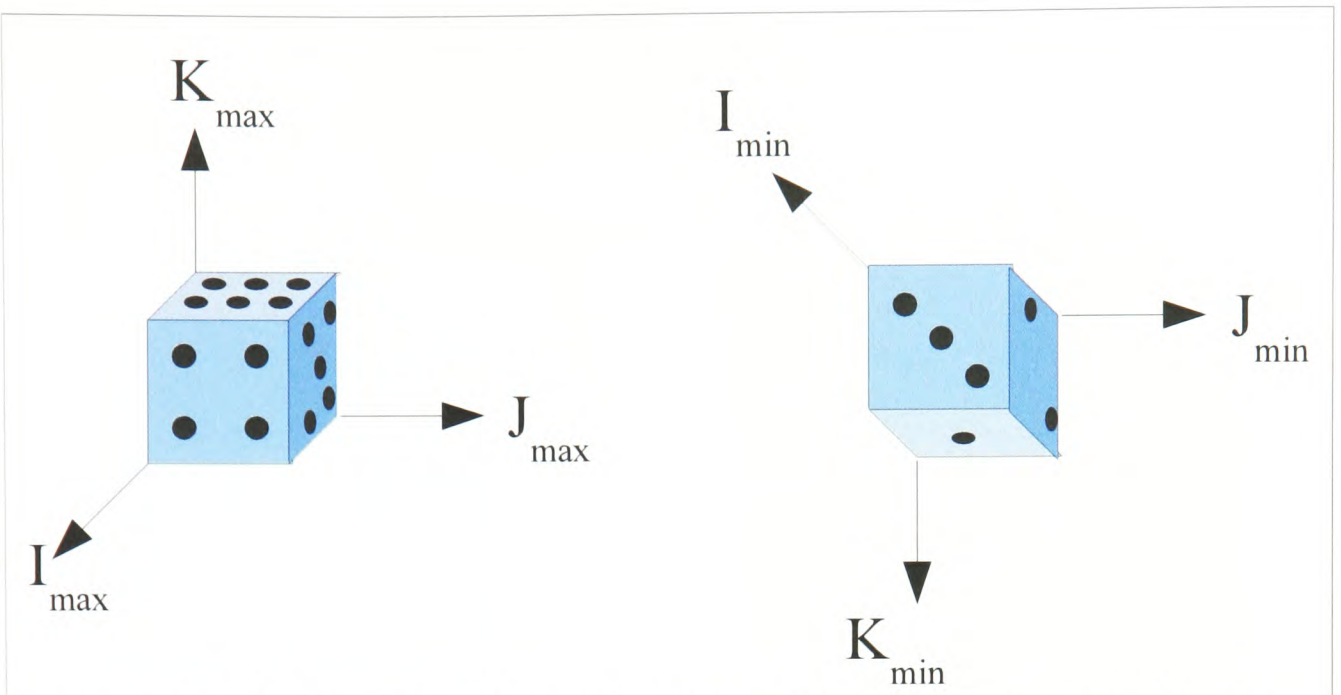


Illustration 4.3.2: Orientation of faces. Coordinates I, J, K are numbered 1,2,3 where coordinates with lower numbers are stored first.

Grid points are stored in such a way that the I direction is treated first, followed by the J and K directions, respectively. This implies that K planes are stored in sequence.

In the following the matching of blocks is outlined. First, it is shown how the orientation of the face of a block is determined. Second, rules are given how to describe the matching of faces between neighboring blocks. This means the determination of the proper orientation values between the neighboring faces. To determine the orientation of a face, arrows are drawn in the direction of increasing coordinate values. The rule is that the lower-valued coordinate varies first, and thereby the orientation is uniquely determined. The orientation of faces between neighboring blocks is determined as follows, see Illustration 4.3.3.Suppose blocks 1 and 2 are oriented as shown. Each individual block has its own coordinate system (right-handed). For example, orientation of block 2 is obtained by rotation of π of block K-axis – rotations are positive in a counterclockwise sense.

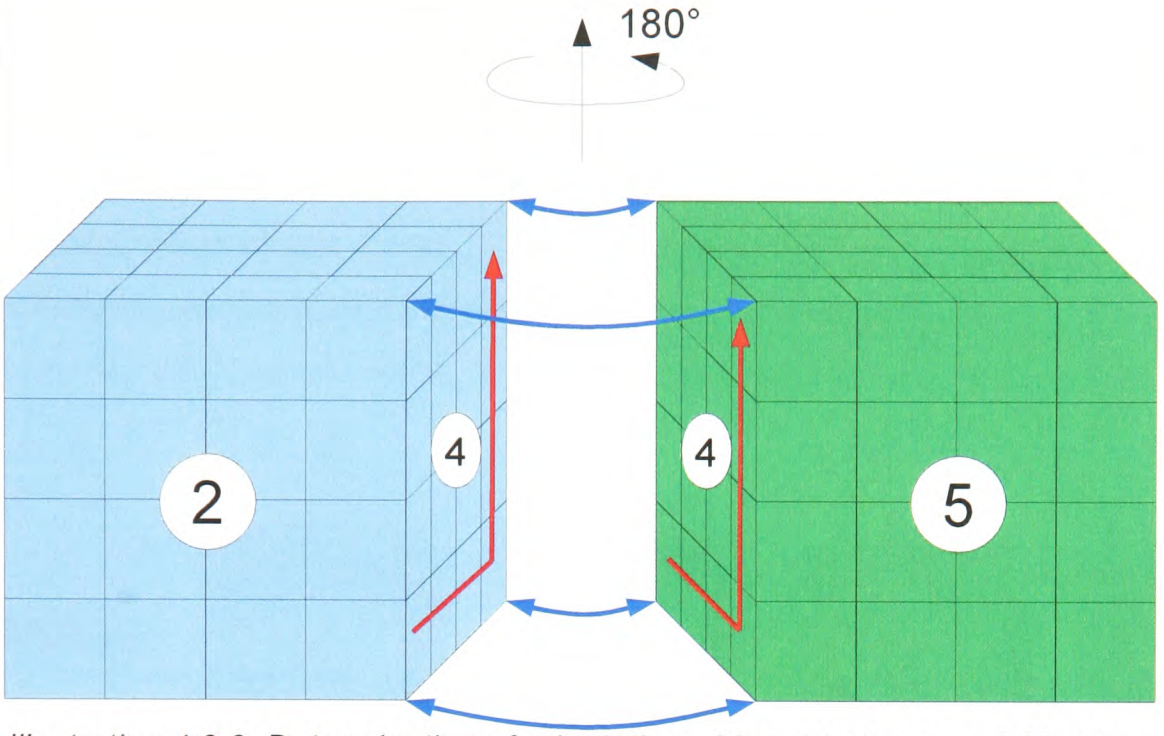


Illustration 4.3.3: Determination of orientation of faces between neighboring blocks as seen from block 1(reference block). The reference block is always oriented as shown and then the corresponding orientation of the neighboring face is determined. (see Illustration 4.3.4)

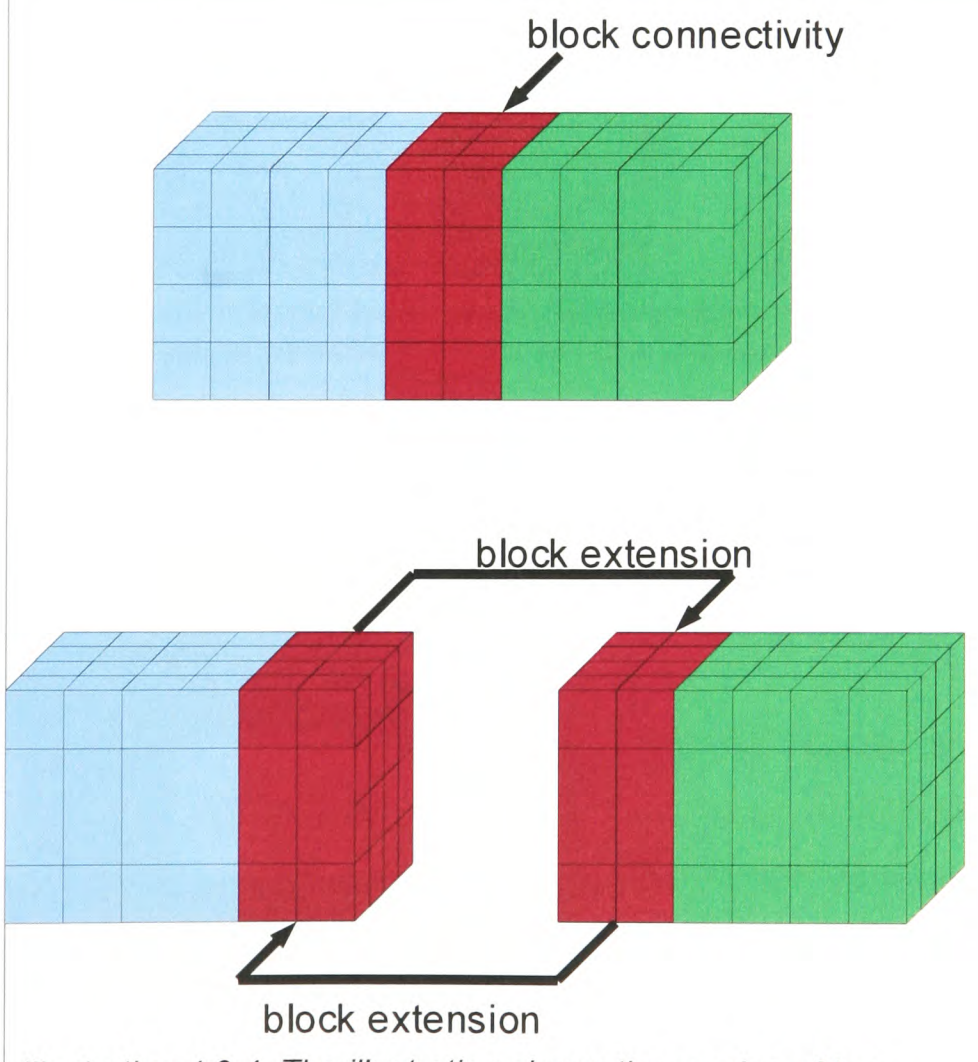


Illustration 4.3.4: The illustration shows the overlap of two neighbouring blocks. For the flow solver, an overlap of two rows or columns is needed. The algorithm is not straightforward, because of the handling of diagonal points.

Thus face 4 of block 1 (used as the reference block) and face 4 of block 2 are matching with the orientations as shown, determined from the rules shown in Illustration 4.3.4. All cases is group 1 can be obtained by rotating a face about an angle of 0, 1/2 π, π or 3/2 π. This is also valid for elements in group 2. The code automatically recognizes when the orientation between two faces needs to be mirrored. Thus cases 1 and 7 in Illustration 4.3.4 are obtained by rotating case 1 by π/2. Here, the rotations are denoted by integers 0,1,2 and 3, respectively.

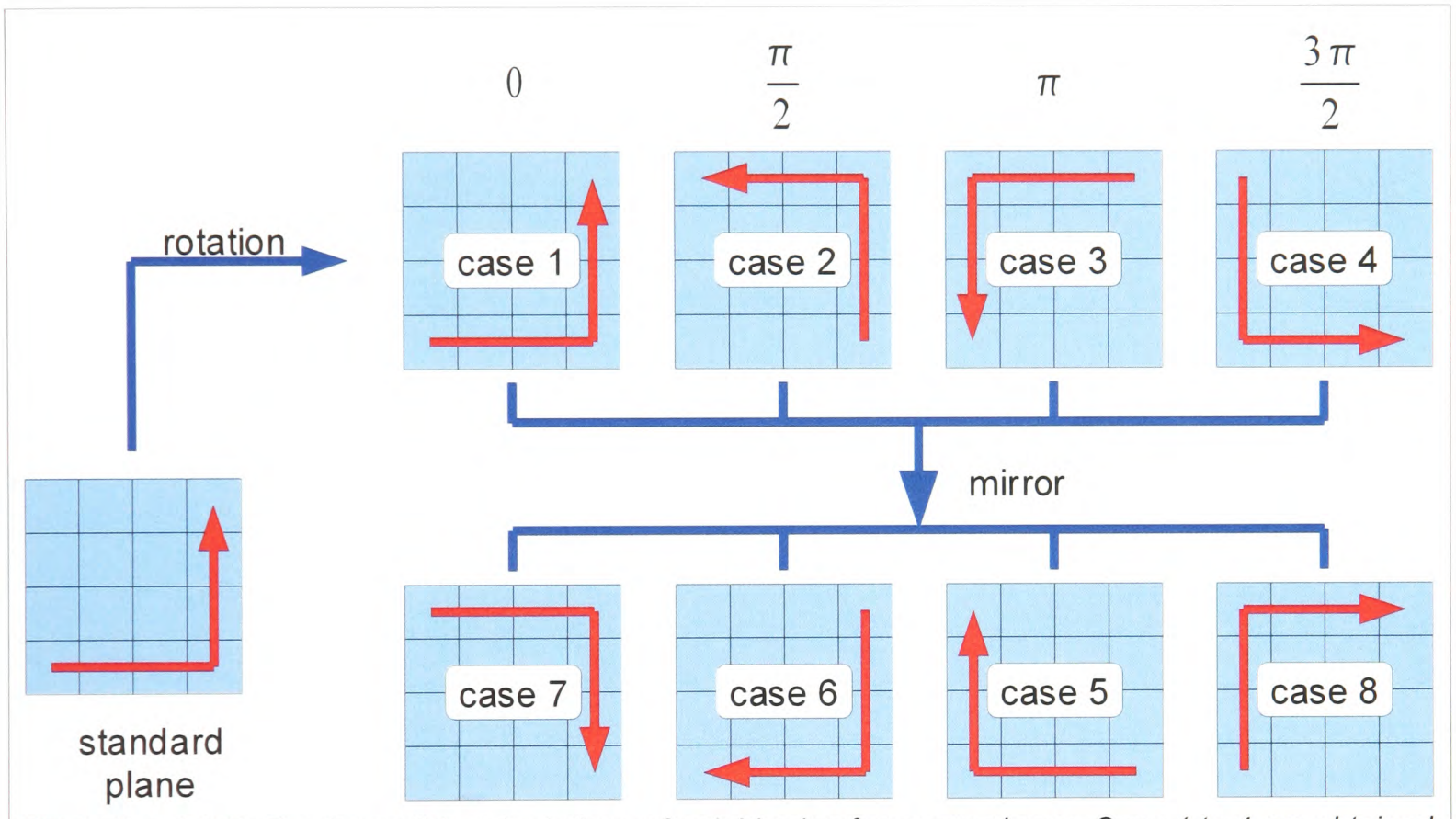


Illustration 4.3.5: The 8 possible orientations of neighboring faces are shown. Case 1 to 4 are obtained by successive rotations. The same situation holds for cases 5 to 8 upon being mirrored.

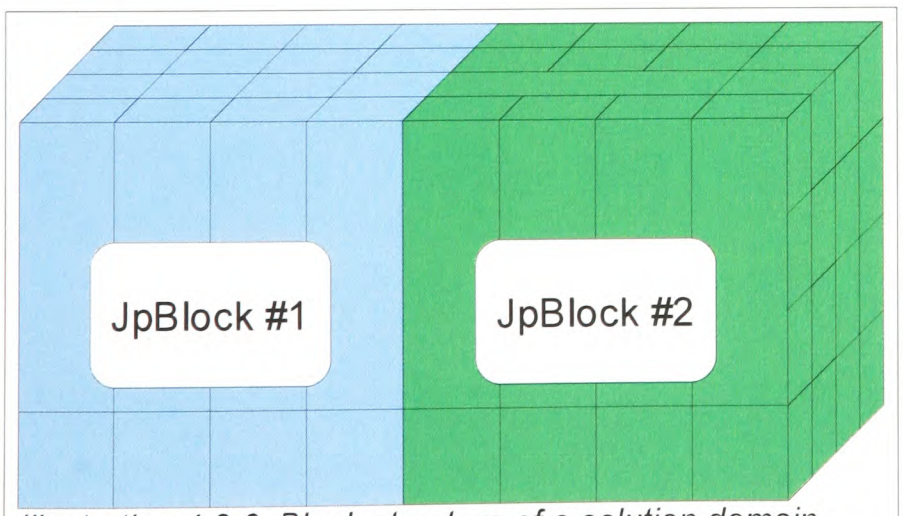


Illustration 4.3.6: Block structure of a solution domain. JUSTGrid creates one JpBlock and one Solver instance per grid block.

4.3.2 JUSTGrid Java class representation of a structured grid

JUSTGrid reads in grid files in various formats see chapter 4.3.2 on page 76. After loading and parsing the grid file JUSTGrid provides a JpBlock array to the compute session. For every grid block one JpBlock- and one solver instance with a unique id will be created.

4 Multiphysics Solver Development with JUSTGrid

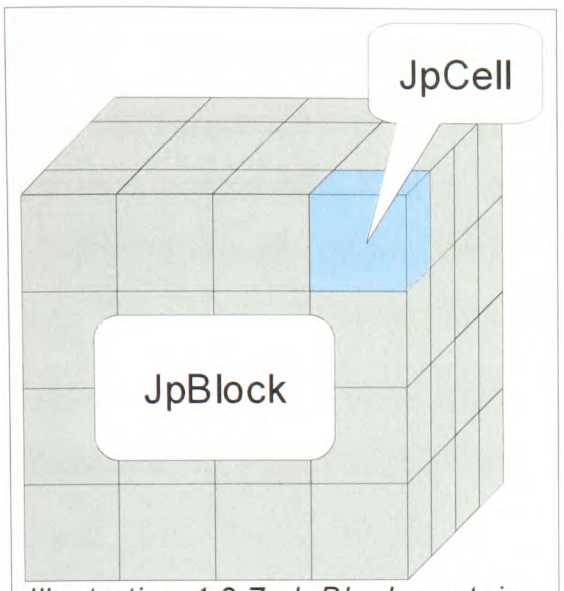


Illustration 4.3.7: JpBlock contains grid data and JpCell instances

A JpBlock contains its unique id the grid data array and an array containing all initialized JpCell instances including all halo cells. The order for both arrays is $[I_{min}\text{-}I_{max}]$ $[J_{min}\text{-}J_{max}]$ $[K_{min}\text{-}K_{max}]$. It is really important to take care of this order while you are writing your own code. Changing the order from [K][J][I] takes much time and raises the possibility of errors.

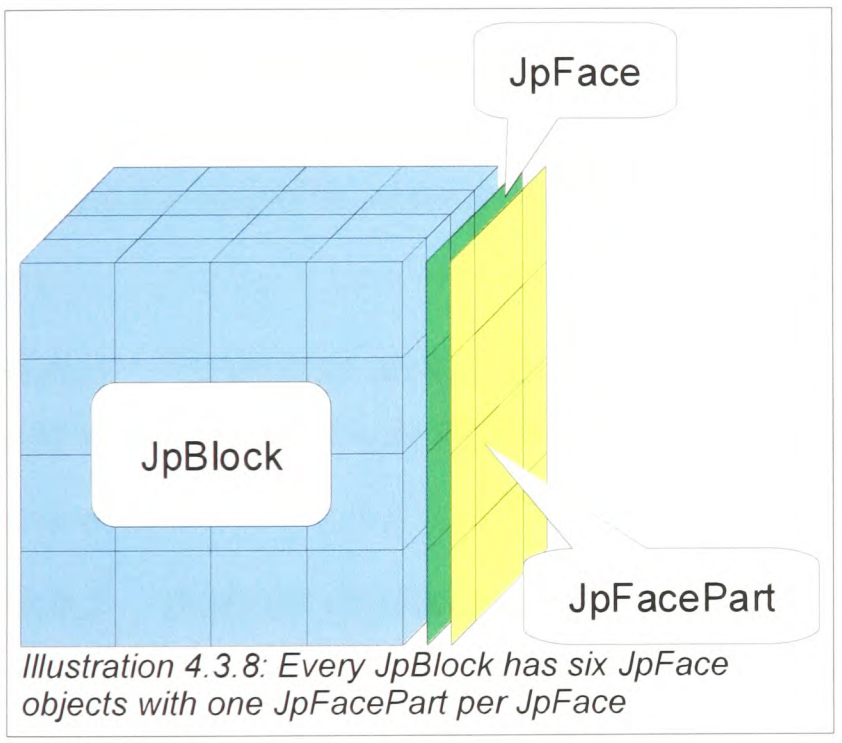


Illustration 4.3.8: Every JpBlock has six JpFace objects with one JpFacePart per JpFace

The connection between block faces and the boundary conditions are stored in the JpFacePart object. Due to the missing implementation of merged blocks every JpFace contains exactly one JpFacePart.

4.4 The `startup.properties` file

The `startup.properties` file contains all information needed for a computation.

4.4.1 Client class section

The client section tells JUSTGrid which classes to use for the current compute session. It is possible to change the default generic solver handler for special initialization methods but normally it is not necessary to do.

```
client.solverhandler.class=hpcc.just.client.JpGenericSolverHandler
client.boundaryhandler.class=simplesolver3d.SimpleBoundaryHandler
client.cell.class=simplesolver3d.SimpleCell
client.solver.class=simplesolver3d.SimpleSolver3D
```

4.4.2 Input and output file section

This section describes all input and output files. JUSTGrid is able to handle more than one input and one output file. This is really important if grid and topology information as well as boundary conditions are stored in different files. JUSTGrid can also store the result using different data types (e.g. Tecplot) in seperate files.

Type and name of each input and output file must be specified. A list follows of all file formats known by JUSTGrid see chapter 4.3.2 on page 76.

```
input.file.type.0=gpg
input.file.name.0=blk.tmp
input.file.type.1=gpc
input.file.name.1=blk.tmp.conn

output.file.type.0=plt
output.file.name.0=output.plt
```

4.4.3 Numerical section

The numerical section contains only one entry, namely the number of halo-cells to be created around the blocks for inter-block connectivity (see Illustration 4.3.2)

```
param.numerical.halocells=1
```

4.4.4 Physical section

The physical section sets the geometry type of the computation 2D or 3D.

```
param.physical.type=3D
```

4.4.5 Solver parameter section

This section can be freely defined by the solver developer or engineer.

```
param.solver.MaxIterations=1000
param.solver.MachNumber=1.0
```

Every solver parameter starts with `solver.param and` will be passed through by JUSTGrid as an initialization value to every solver instance. The technique to communicate between the `startup.properties` file and the solver instance is very easy for the deveploper. Simply write a name corresponding to the so called setter method into the solver class. In our case this would be:

```
public class SimpleSolver3D extends JpMultiblockSolver
{
  public setMaxIterations( int maxIteration )
  {
    ...
  }

  public setMachNumber( double mach )
  {
    ...
```

```
  }
...
```

Be careful to note that JUSTGrid is case sensitive dealing when with method names. During the method recognition for the solver parameters, JUSTGrid will follow this sequence:

```
setMethod( double v), setMethod( int v), setMethod( String s)
```

If a matching method is found JUSTGrid invokes this method on all solver instances and continues with the next parameter.

5 Multiphysics Equations in *JUSTGrid*

As an example of a nontrivial system of multiphysics equations the magneto-hydrodynamics (MHD) equations were chosen. These equations are a combination of the nonlinear equations of fluid dynamics, described by the Navier-Stokes equations and Maxwell's equations of electrodynamics, and thus represent a genuine multiphysics problem. In addition, the numerical solution of these equations exhibits unique challenges in the form of magnetoacoustic and Alfvén waves. Moreover, the constraint of $\nabla \cdot \boldsymbol{B}=0$ is difficult to maintain. In addition, it must be ensured that any initial solution numerically satisfies this constraint. In contrast to the analytic solution, which remains divergence free, the numerical solution needs special treatment to guarantee this feature. This combination of fluid- and electrodynamics, having a wide range of applications (plasma physics, aero- and aerothermodynamics, fusion, astrophysics, gas discharges etc.), requires a challenging numerical solution procedure, because waves from both fluid dynamics and electrodynamics are present and must be properly resolved.

5.1 Introduction

MHD is useful, if charge separation is negligible. Length scales need to be larger than the Debye length, and time scales larger than the inverse of the plasma frequency. In other words, the model cannot be applied to high-frequency phenomena that apply large separation. In order to guarantee isotropy, the collision frequency has to be higher than the cyclotron frequency.

To further simplify the equations, it should be noted that the displacement current can be neglected, because in $\nabla \times \boldsymbol{B}=\frac{4\pi}{c}\boldsymbol{j}+\frac{1}{c}\frac{\partial \boldsymbol{E}}{\partial t}$, $\frac{\frac{1}{c}\frac{\partial \boldsymbol{E}}{\partial t}}{\nabla \times \boldsymbol{B}} \approx \frac{v^2}{c^2}$ and thus the time derivative of the electric field can be neglected.

It should be noted that from now on the Maxwell equations will be written exclusively using the SI system, which is more suitable for engineering purposes. Using the Maxwell equations in the MKS System (which is being used throughout this thesis), $\nabla \times \boldsymbol{B}=\mu_0 \boldsymbol{j}=\mu_0 \sigma(\boldsymbol{E}+\boldsymbol{v}\times\boldsymbol{B})$ and thus $\boldsymbol{E}=\frac{1}{\mu_0 \sigma}\nabla \times \boldsymbol{B}-\boldsymbol{v}\times\boldsymbol{B}$ is a dependent variable, and therefore electric field strength $\boldsymbol{E}$ is not computed in MHD. That is, the corresponding Maxwell equation is not needed.

5.2 *Magnetohydrodynamic (MHD) Equations*

5.2.1 MHD Equations

The MHD equations are the combination of Navier-Stokes and Maxwell equations together with Ohm's law. The governing equations are listed below.

Continuity equation:

$$\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho \boldsymbol{v})=0 \tag{5.2.1}$$

Momentum equation:

$$\frac{\partial(\rho \boldsymbol{v})}{\partial t}+\nabla\cdot[\rho \boldsymbol{v}\boldsymbol{v}-\frac{\boldsymbol{B}\boldsymbol{B}}{\mu_m}+P\,I]-\frac{1}{Re}\nabla\cdot\boldsymbol{\tau}=0 \tag{5.2.2}$$

It should be noted that the expression ***BB*** in Eqs. (5.2.2) stands for a second rank tensor. Similar terms occur in Eqs. (5.2.4) where the order of the factors is important (***vB*** and ***Bv***).

Energy equation:

$$\begin{aligned}&\frac{\partial(\rho \boldsymbol{E})}{\partial t}+\nabla\cdot[(\rho E+P)\boldsymbol{v}-\boldsymbol{B}(\frac{\boldsymbol{v}\cdot\boldsymbol{B}}{\mu_m})\\&-\frac{1}{Re}(\boldsymbol{v}\cdot\boldsymbol{\tau})-\frac{1}{(\gamma-1)P_r Ma^2 Re}\dot{\boldsymbol{q}}-(\frac{\boldsymbol{B}}{\mu_m\sigma}\cdot\frac{\nabla\boldsymbol{B}}{\mu_m}-\nabla\frac{B^2}{\mu_m^2\sigma})]=0\end{aligned} \tag{5.2.3}$$

Induction equation:

$$\frac{\partial \boldsymbol{B}}{\partial t}+\nabla\cdot(\boldsymbol{v}\boldsymbol{B}-\boldsymbol{B}\boldsymbol{v})+\nabla\times[\frac{1}{\sigma}(\nabla\times\frac{\boldsymbol{B}}{\mu_m})]=0 \tag{5.2.4}$$

where $P=p+\frac{B^2}{2\mu_m}$, $E=\frac{p}{(\gamma-1)\rho}+\frac{v^2}{2}+\frac{B^2}{2\mu_m\rho}$, and $B^2=\boldsymbol{B}\cdot\boldsymbol{B}$, $v^2=\boldsymbol{v}\cdot\boldsymbol{v}$.

Here E is the total energy per mass unit, comprising *internal*, *kinetic*, and *magnetic* energies. P is the total pressure, p the static pressure, $\boldsymbol{\tau}$ denotes stress tensor, and, in terms of temperature T, the heat flux vector is given by $\dot{\boldsymbol{q}}=k\nabla T$. The equations have to be supplemented by models for conductivity ***s*** and magnetic permeability μ_m.

5.2.2 Ideal MHD Equations

The classic ideal magneto-hydrodynamics (MHD) governing equations can be deduced from the MHD system given above with additional assumptions. First, the concept of infinite electrical conductivity implies that the strength of the motion-induced magnetic field overwhelms that of the applied field. Second, in many flows inertial effects greatly outweigh viscous dissipation and heat transfer in the medium. Third, the medium is considered to be isotropic (see [SHA01]). Then the resulting equations are:

$$\frac{\partial \boldsymbol{U}}{\partial t}+\nabla\cdot\boldsymbol{F}=0 \tag{5.2.5}$$

where F is a second rank tensor and $\boldsymbol{U}$ is the vector of variables given by

$$\boldsymbol{U}=\begin{bmatrix}\rho\\ \rho\,\boldsymbol{v}\\ E\\ \boldsymbol{B}\end{bmatrix} \tag{5.2.6}$$

$$\boldsymbol{F}=\begin{bmatrix}\rho\,\boldsymbol{v}\\ \rho\,\boldsymbol{v}\,\boldsymbol{v}+P\,\boldsymbol{I}-\boldsymbol{B}\,\boldsymbol{B}\\ (E+P)\boldsymbol{v}-\boldsymbol{B}(\boldsymbol{v}\cdot\boldsymbol{B})\\ \boldsymbol{v}\,\boldsymbol{B}-\boldsymbol{B}\,\boldsymbol{v}\end{bmatrix} \tag{5.2.7}$$

where ρ is mass density, $\boldsymbol{v}=(u,v,w)^T$ is the velocity, $\boldsymbol{B}=(B_x,B_y,B_z)^T$ is the magnetic induction field, where E now is the total energy per unit volume, which is defined as (for ideal MHD)

$$E=p/(\gamma-1)+\rho(u^2+v^2+w^2)/2+(B_x^2+B_y^2+B_z^2)/2\,\mu_m \tag{5.2.8}$$

and total pressure is

$$P=p+(B_x^2+B_y^2+B_z^2)/2\,\mu_m. \tag{5.2.9}$$

In additional to the above equations, the magnetic field satisfies the divergence free constraint $\nabla\cdot\boldsymbol{B}=0$. This is not an evolution equation and has to be satisfied numerically at each iteration

step for any kind of grid. Special care has to be taken to guarantee that this condition is satisfied, otherwise the solution may become non-physical. Due to the coupling of the induction equation to the momentum and energy equations, these quantities would also be modeled incorrectly. A special problem arises to guarantee this condition satisfied at curved boundaries.

5.3 MHD Waves

The above ideal MHD equations constitute a non-strictly hyperbolic partial differential system [SHA02]. From the analysis of the governing equations in one-dimensional spatio-temporal space, eigenvector and eigenvalues have been found. The remaining seven eigenvalues of the MHD equations can also locally degenerate to coincide with each other, depending on the relative magnitude and orientation of the magnetic field. The seven eigenvalues are : $[u, u \pm c_A, u \pm c_s, u \pm c_f]$. All velocity components are in the direction of propagation of the wave.

These eigenvalues reflect four different wave speeds for a perturbation propagating in a plasma field: the usual acoustic, the Alfven , and the slow and fast plasma waves:

$$a^2 = \left(\frac{\partial p}{\partial \rho}\right)_s \tag{5.3.1}$$

$$c_A^2 = B_n^2 / \rho \mu_m \tag{5.3.2}$$

where B_n denotes the transverse (normal) component of the magnetic induction field with respect to the wave front.

$$2c_s^2 = a^2 + \frac{B^2}{\rho \mu_m} - \sqrt{\left(a^2 + \frac{B^2}{\rho \mu_m}\right)^2 - 4a^2 c_A^2} \tag{5.3.3}$$

$$2c_f^2 = a^2 + \frac{B^2}{\rho \mu_m} + \sqrt{\left(a^2 + \frac{B^2}{\rho \mu_m}\right)^2 - 4a^2 c_A^2} \tag{5.3.4}$$

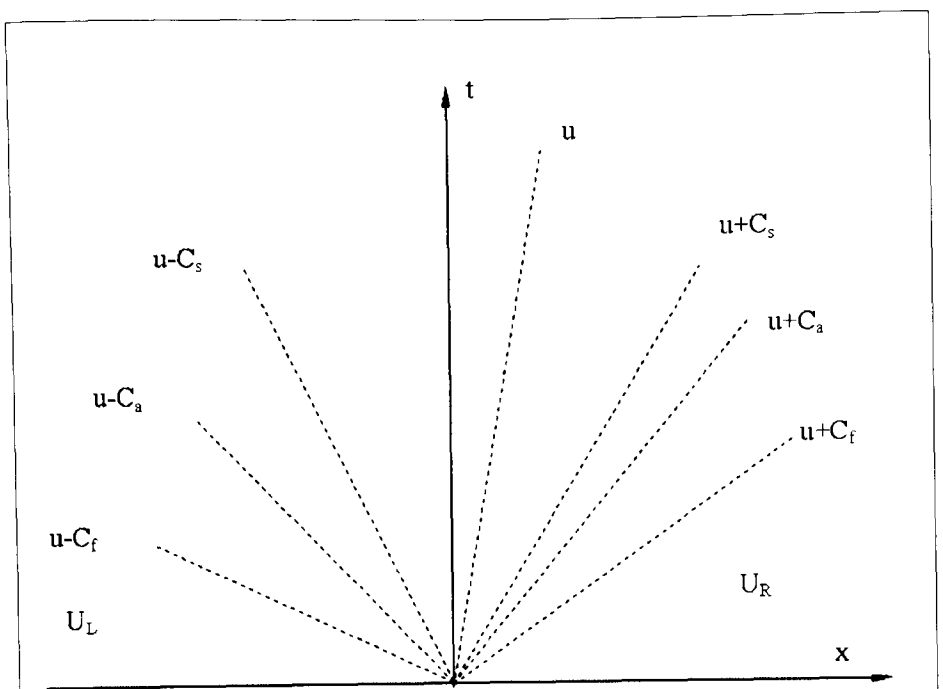


Illustration 5.3.1: Waves in a 1-D MHD Riemann problem.

Illustration 5.5.1 shows the {x-t} diagram of all waves at a cell interface resulting from the linearized ideal MHD equations.

5.4 *Flux Formulation using the HLLC Riemann Solver*

First, we consider the HLLC (Harten-Lax-van Leer-Contact discontinuity) scheme for the Euler equations only. Then we extend the HLLC scheme to the MHD equations. The HLLC scheme is developed from the HLL scheme.

5.4.1 HLL Flux Formulation

If we consider the shock tube problem, we encounter three different types of waves, namely a shock wave, a contact discontinuity (across which only temperature and density vary) and a rarefaction wave. If the initial conditions are such that the shock wave and the contact discontinuity move to the right, the rarefaction wave is moving to the left. A diaphragm may separate two states of variables in the shock tube, denoted as left and right states. Accordingly, all variables are indexed with the letters *l* and *r.* Across the diaphragm, thought to be of zero thickness, variables are discontinuous. Depending on the values of the left and right states, various flow scenarios may develop. The shock tube is thought to be of infinite length, and variables vary only in the direction of the flow. Flow is uniform in the lateral direction. In this respect, the shock tube is a model for the Riemann problem in one dimension. The Riemann problem consists of the PDE and the initial conditions (IC). There are no boundary conditions (BC) since the region is not bounded.

$$\boldsymbol{U}_t+\boldsymbol{F}_x(\boldsymbol{U})=0;\quad IC:\ \boldsymbol{U}(x,0)=\begin{cases}\boldsymbol{U}_l if\ x<0\\ \boldsymbol{U}_r if\ x>0\end{cases} \tag{5.4.1}$$

The initial values for fluxes are denoted in a similar way,

$$IC: \quad \boldsymbol{F}(\boldsymbol{U}(x,0)) = \begin{Bmatrix} \boldsymbol{F}_l = \boldsymbol{F}(\boldsymbol{U}_l) if\ x<0 \\ \boldsymbol{F}_r = \boldsymbol{F}(\boldsymbol{U}_r) if\ x>0 \end{Bmatrix} \tag{5.4.2}$$

Numerically, values of $\boldsymbol{U}$ are known at cell centers only, indicated by index *i* (one-dimensional problem), but fluxes need to be computed at cell faces with index $i+1/2$, and thus approximations to the flux function $\boldsymbol{F}_{i+1/2}$ are to be found. Here, the approach by Harten, Lax, and van Leer (HLL) is followed, with corrections implemented by Batten] to account for the contact discontinuity (hence the scheme is termed HLLC).

The derivation of the flux function is performed in two stages. First, the HLL scheme is derived, and in the second stage, the scheme is modified to incorporate the contact discontinuity, producing the HLLC scheme. A major task is the evaluation of the wave propagation speeds. If u, a denote the flow speed and the speed of sound, respectively, the Riemann problem has 3 distinct eigenvalues, namely $u-a, u, u+a$ where the u eigenvalue has multiplicity 3. In order to approximate the flux function, the above specified Riemann problem is solved on the domain (x_l, x_r) and integrated in time from $0\ to\ t_f$. One obtains

$$\int_{x_l}^{x_r} \boldsymbol{U}(x,t_f)\,dx = \int_{x_l}^{x_r} \boldsymbol{U}(x,0)\,dx + \int_0^{t_f} \boldsymbol{F}(\boldsymbol{U}(x_l,t))\,dt - \int_0^{t_f} \boldsymbol{F}(\boldsymbol{U}(x_r,t))\,dt. \tag{5.4.3}$$

In order to evaluate the integrals, the (yet unknown) signal speeds $s_l and\ s_r$ are considered, denoting the fastest wave propagation in the negative and positive x-directions. It is assumed, however, that at the final time t_f, no information has reached the left, $x_l<0$, and right, $x_r>0$, boundaries of the spatial integration interval, that is

$$x_l < s_l t_f \ and\ x_r > s_r t_f \tag{5.4.4}$$

Under this assumption, we obtain $\boldsymbol{U}(x_l,t_f)=\boldsymbol{U}(x_l,0) and \boldsymbol{U}(x_r,t_f)=\boldsymbol{U}(x_r,0)$. Therefore the last two integrals on the RHS can be immediately evaluated. Using the initial conditions, the first integral on the RHS is easily calculated, resulting in

$$\int_{x_l}^{x_r} \boldsymbol{U}(x,t_f)dx = x_r\boldsymbol{U}_r - x_l\boldsymbol{U}_l + t_f(\boldsymbol{F}_r - \boldsymbol{F}_l) \tag{5.4.5}$$

At time t_f waves have moved according to their propagation speeds and information has been carried along the x-axis. Naturally, the initial solution has changed. For the time being, we only consider two waves with speeds s_l and s_r. Since $s_r > s_l$, the x domain is subdivided into three intervals, namely $(x_l, s_l t_f), (s_l t_f, s_r t_f), and (s_r t_f, x_r).$ $\boldsymbol{U}$ is constant within each interval, but may be discontinuous across each wave (characteristic curve). The integral on the LHS of Eq. (5.4.5) therefore has to be split into 3 integrals. Since no information has reached the first and the third intervals, these integrals can be directly calculated.

$$\int_{s_l t_f}^{s_r t_f} \boldsymbol{U}(x,t_f)dx = -\int_{x_l}^{s_l t_f} \boldsymbol{U}(x,t_f)dx - \int_{s_r t_f}^{x_r} \boldsymbol{U}(x,t_f)dx + x_r\boldsymbol{U}_r - x_l\boldsymbol{U}_l + t_f(\boldsymbol{F}_r - \boldsymbol{F}_l) \tag{5.4.6}$$

Evaluating the integral it follows that

$$\boldsymbol{U}(x,t_f) = \boldsymbol{U}_l \; for \; x \in (x_l, s_l t_f) and \; \boldsymbol{U}(x,t_f) = \boldsymbol{U}_r \; for \; x \in (s_l t_f, x_r) \tag{5.4.7}$$

Inserting these values results in

$$\int_{s_l t_f}^{s_r t_f} \boldsymbol{U}(x,t_f)dx = (x_l - s_l t_f)\boldsymbol{U}_l + (s_r t_f - x_r)\boldsymbol{U}_r + x_r\boldsymbol{U}_r - x_l\boldsymbol{U}_l + t_f(\boldsymbol{F}_r - \boldsymbol{F}_l) \tag{5.4.8}$$

or

$$\int_{s_l t_f}^{s_r t_f} \boldsymbol{U}(x,t_f)dx = s_r t_f \boldsymbol{U}_r - s_l t_f \boldsymbol{U}_l + t_f(\boldsymbol{F}_r - \boldsymbol{F}_l) \tag{5.4.9}$$

With the definition

$$\boldsymbol{U}_{HLL} := \frac{1}{t_f(s_r - s_l)} \int_{s_l t_f}^{s_r t_f} \boldsymbol{U}(x, t_f)\, dx = \frac{s_r \boldsymbol{U}_r - s_l \boldsymbol{U}_l + \boldsymbol{F}_r - \boldsymbol{F}_l}{s_r - s_l} \tag{5.4.10}$$

Where $\boldsymbol{U}_{HLL}$ is a constant, depending on the hitherto unknown wave speeds. For a given time $t \in (0, t_f)$, the Riemann solution can thus be written in the form

$$\boldsymbol{U}(x,t) = \begin{cases} \boldsymbol{U}_L & if\ x < s_l t \\ \boldsymbol{U}_{HLL} & if\ s_l t < x < s_l t \\ \boldsymbol{U}_R & if\ x > s_r t \end{cases} \tag{5.4.11}$$

The disadvantage of this solution is that contact discontinuities are not properly accounted for that is, all intermediate states that might exist in the region $(s_l t_f, s_r t_f)$ were averaged over by the integration process. Hence, if the solution contains a contact discontinuity, it has been smeared out, and will not be present in the numerical solution.

The numerical flux computation for the supersonic case is straightforward. Information is traveling only in one direction, and thus the time dependent flux $\boldsymbol{F}_{HLL}$ at surface $x=0$ (denoted by $\boldsymbol{F}_{i+1/2}$ in the finite volume approach through the face labeled $i+1/2$) is either the flux $\boldsymbol{F}_l$ *or flux* $\boldsymbol{F}_r$. We therefore need only to consider the subsonic case where information across surface $x=0$ (Riemann problem) can arrive both sides, namely from the upwind and the downwind directions. In that case $s_l < 0 < s_r$. For instance, if $u > 0$ (flow velocity) then $s_l = u - a$ *and* $s_r = u + a$. The question arises how to compute flux $\boldsymbol{F}_{HLL}$. One immediate possibility is to set $\boldsymbol{F}_{HLL} = \boldsymbol{F}(\boldsymbol{U}_{HLL})$. However, flux is an integral quantity, and using an averaged state vector instead of an averaged flux may not be a good approximation. In any case, this has nothing to do with conservation laws, it would be a purely mathematical procedure, and therefore is not conservative. In the general case the interface flux $\boldsymbol{F}_{HLL}$ becomes

$$\boldsymbol{F}_{HLL} = \begin{cases} \boldsymbol{F}_l & if\ 0 < s_l \\ \boldsymbol{F}^*_{lr} & if\ s_l \le 0 \le s_r \\ \boldsymbol{F}_r & if\ s_r < 0 \end{cases} \tag{5.4.12}$$

and with

$$\boldsymbol{F}_{lr}^{*}=\frac{s_r \boldsymbol{F}_l - s_l \boldsymbol{F}_r + s_l s_r (\boldsymbol{U}_r - \boldsymbol{U}_l)}{s_r - s_l} \tag{5.4.13}$$

5.4.2 HLLC Flux

The HLLC flux is a modification of the HLL flux. Instead of a single intermediate state $\boldsymbol{U}_{HLL}$ two intermediate states $\boldsymbol{U}_l^* \; and \; \boldsymbol{U}_r^*$ are assumed, separated by an interface moving with speed s_M [BAT01]:

$$\boldsymbol{U}_{HLLC}=\begin{cases} \boldsymbol{U}_l & if\ 0<s_l \\ \boldsymbol{U}_l^* & if\ s_l \le 0 \le s_M \\ \boldsymbol{U}_r^* & if\ s_M \le 0 \le s_r \\ \boldsymbol{U}_r & if\ s_r<0 \end{cases} \tag{5.4.14}$$

The corresponding interface flux denoted $\boldsymbol{F}_{HLLC}$, is defined as

$$\boldsymbol{F}_{HLLC}=\begin{cases} \boldsymbol{F}_l & if\ 0<s_l \\ \boldsymbol{F}_l^* & if\ s_l \le 0 \le s_M \\ \boldsymbol{F}_r^* & if\ s_M \le 0 \le s_r \\ \boldsymbol{F}_r & if\ s_r<0 \end{cases} \tag{5.4.15}$$

Applying the Rankine-Hugoniot conditions:

$$\boldsymbol{F}_l^* = \boldsymbol{F}_l + s_l(\boldsymbol{U}_l^* - \boldsymbol{U}_l) \tag{5.4.16}$$

and

$$\boldsymbol{F}_r^* = \boldsymbol{F}_r + s_r(\boldsymbol{U}_r^* - \boldsymbol{U}_r) \tag{5.4.17}$$

to determine values $\boldsymbol{U}_l^* \; and \; \boldsymbol{U}_r^*$ Batten [] made the assumption that

$$s_M = q_l^* = q_r^* = q^* \tag{5.4.18}$$

and got the following results

$$s_M = \frac{\rho_r q_r (s_r - q_r) - \rho_l q_l (s_l - q_l) + p_l - p_r}{\rho_r (s_r - q_r) - \rho_l (s_l - q_l)} \tag{5.4.19}$$

$$\rho_k^* = \rho_k \frac{s_k - q_k}{s_k - s_M} \tag{5.4.20}$$

$$p^* = \rho_k (q_k - s_k)(q_k - s_M) + p_k \tag{5.4.21}$$

$$(\rho u)_k^* = \frac{(s_k - q_k)\rho_k u_k + (p^* - p_k) n_x}{s_k - s_m} \tag{5.4.22}$$

$$(\rho v)_k^* = \frac{(s_k - q_k)\rho_k v_k + (p^* - p_k) n_y}{s_k - s_m} \tag{5.4.23}$$

$$(\rho w)_k^* = \frac{(s_k - q_k)\rho_k w_k + (p^* - p_k) n_y}{s_k - s_m} \tag{5.4.24}$$

$$e_k^* = \frac{(s_k - q_k) e_k - p_k q_k + p^* s_M}{s_k - s_M} \tag{5.4.25}$$

In Eqs. from 5.4.20 to 5.4.25, the subscript *k* stands for *l* or *r*. Using Eqs. 5.4.19 to 5.4.25, the flux can be calculated as following:

$$\boldsymbol{F}_k = \begin{Bmatrix} \rho_k^* q_k^* \\ \rho_k^* u_k^* q_k^* + p^* n_x \\ \rho_k^* v_k^* q_k^* + p^* n_y \\ \rho_k^* w_k^* q_k^* + p^* n_z \\ (e_k^* + p^*) q_k^* \end{Bmatrix} \tag{5.4.26}$$

5.4.3 HLLC for Magneto-Gasdynamics Equations (MHD-HLLC)

5.4.3.1 Derivation of MHD-HLLC Riemann Solver

Now, the HLLC scheme for MHD can be derived. The 2-D MHD equations. are considered. Rewriting Eqs. 5.4.16 and 5.4.17 for the MHD equations, results in (here the subscripts l and r are dropped for simplicity)Error: Reference source not found.

$$s\begin{bmatrix}\rho^*\\ \rho^* u^*\\ \rho^* v^*\\ B_x^*\\ B_y^*\\ B_z^*\\ E^*\end{bmatrix}-\begin{bmatrix}\rho^* q^*\\ \rho^* u^* q^* + P^* n_x - B_n^* B_x^*\\ \rho^* v^* q^* + P^* n_y - B_n^* B_y^*\\ q^* B_x^* - B_n^* u^*\\ q^* B_y^* - B_n^* v^*\\ 0\\ (E^*+P^*)q^* - B_n^*(\boldsymbol{B}\cdot\boldsymbol{v})^*\end{bmatrix}=s\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ B_x\\ B_y\\ B_z\\ E\end{bmatrix}-\begin{bmatrix}\rho q\\ \rho u q + P n_x - B_n B_x\\ \rho v q + P n_y - B_n B_y\\ q B_x - B_n u\\ q B_y - B_n v\\ 0\\ (E+P)q - B_n(\boldsymbol{B}\cdot\boldsymbol{v})\end{bmatrix} \tag{5.4.27}$$

where $B_n = B_x n_x + B_y n_y$ *and* $q = u n_x + v n_y$. Similar to Eq. 5.4.19, the speed q^* for MHD can be assumed as $S_M = q_l^* = q_r^* = q_l^*$ and can be obtained from the HLL approximation (Eq. Error: Reference source not found). This results in the following expression:

$$q^* = s_M = \frac{\rho_r q_r (s_r - q_r) - \rho_l q_l (s_l - q_l) + p_l - p_r - B_{nl}^2 + B_{nr}^2}{\rho_r (s_r - q_r) - \rho_l (s_l - q_l)} \tag{5.4.28}$$

in order to make the HLLC middle state $\boldsymbol{U}^*$ consistent with the integral form of the conservation laws, which is described as consistency condition by Toro.

$$\frac{q^* - s_l}{s_r - s_l}\boldsymbol{U}_l^* + \frac{s_r - q^*}{s_r - s_l}\boldsymbol{U}_r^* = \frac{s_r \boldsymbol{U}_r - s_l \boldsymbol{U}_l - (F_r - F_l)}{s_r - s_l} \tag{5.4.29}$$

Shengtai Li suggests:

$$B_{xl}^* = B_{xr}^* = B_x^{HLL} = \frac{s_r B_{xr} - s_l B_{xl}}{s_r - s_l} \tag{5.4.30}$$

and

$$B_{yl}^* = B_{yr}^* = B_y^{HLL} \quad , \quad B_{zl}^* = B_{zr}^* = B_z^{HLL} \tag{5.4.31}$$

$$B_{xl}^*(\boldsymbol{B}\cdot\boldsymbol{v})_l^* = B_{xr}^*(\boldsymbol{B}\cdot\boldsymbol{v})_r^* \quad or \quad (\boldsymbol{B}\cdot\boldsymbol{v})_l^* = (\boldsymbol{B}\cdot\boldsymbol{v})_r^* \ . \tag{5.4.32}$$

$$(\boldsymbol{B}\cdot\boldsymbol{v})_l^* = (\boldsymbol{B}\cdot\boldsymbol{v})_r^* := \boldsymbol{B}^{HLL}\cdot\boldsymbol{v}^{HLL} \tag{5.4.33}$$

and then

$$P^* = \rho(s_l - q)(q^* - q) + P - B_n^2 + (B_n^*)^2. \tag{5.4.34}$$

With the known values of $B_x^*, B_y^*, B_z^*, q^*, and\ P^*$ the rest of the components can be derived easily:

$$\begin{aligned}
\rho_K^* &= \rho_K \frac{S_K - q_K}{S_K - q^*} \\
(\rho u)_K^* &= (\rho u)_K \frac{S_K - q_K}{S_K - q^*} + \frac{(P^* - P_K)n_x + B_{nK}B_{xK} - B^* B_x^*}{S_K - q^*} \\
(\rho v)_K^* &= (\rho v)_K \frac{S_K - q_K}{S_K - q^*} + \frac{(P^* - P_K)n_y + B_{nK}B_{yK} - B^* B_y^*}{S_K - q^*} \\
E_K^* &= E_K \frac{S_K - q_K}{S_K - q^*} + \frac{P^* q^* - P_K q_K + B_{nK}(\boldsymbol{B}\cdot\boldsymbol{v})_K - B^*(\boldsymbol{B}\cdot\boldsymbol{v})^*}{S_K - q^*}.
\end{aligned} \tag{5.4.35}$$

The quantity $\boldsymbol{v}^{HLL}$ can be calculated from the conservative variables. We remark that if we had chosen $(\boldsymbol{B}\cdot\boldsymbol{v})_k^* = \boldsymbol{B}_k^*\cdot\boldsymbol{v}_k^*$, Eq. 5.4.29 would not be satisfied by the given expressions of $\boldsymbol{B}^*\ and\ \boldsymbol{V}^*$.

We can now write the MHD-HLLC flux as

$$\boldsymbol{F}_{HLLC} = \begin{cases} \boldsymbol{F}_l & if\ 0 < s_l \\ \boldsymbol{F}_l^* = \boldsymbol{F}_l + s_l(\boldsymbol{U}_l^* - \boldsymbol{U}_l) & if\ s_l \le 0 \le q^* \\ \boldsymbol{F}_r = \boldsymbol{F}_r + s_r(\boldsymbol{U}_r^* - \boldsymbol{U}_r)^* & if\ q^* \le 0 \le s_r \\ \boldsymbol{F}_r & if\ s_r < 0 \end{cases}. \tag{5.4.36}$$

5.4.3.2 Summary of the Formulas for Two-dimensional Ideal MHD-HLLC

For the two-dimensional ideal MHD equations (Eq. 5.2.5), the following formulas are used:

- Formulas for flux calculation

Define

$$\begin{aligned} \boldsymbol{B} &= (B_x, B_y, B_z)^T \\ \boldsymbol{v} &= (u, v)^T \\ B_n &= B_x n_x + B_y n_y \\ v_n &= u n_x + v n_y \end{aligned} \tag{5.4.37}$$

where index n denotes normal direction.

The flux is given as

$$\boldsymbol{F} = \begin{pmatrix} \rho v_n \\ \rho v_n u + P n_x - B_n B_x \\ \rho v_n v + P n_y - B_n B_y \\ (E+P) v_n - B_n (u B_x + v B_y) \\ v_n B_x - B_n u \\ v_n B_y - B_n v \\ 0 \end{pmatrix}. \tag{5.4.38}$$

- *Wavespeed Formulas*

  The formulas for calculating wavespeeds are given by Eqs. 5.3.1 to 5.3.4.

For all dependent variables Roe-averaged values are used:

$$\begin{aligned} \tilde{\rho} &= \sqrt{\rho_l \rho_r} \\ \tilde{u} &= \frac{u_l \sqrt{\rho_l} + u_r \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \\ \tilde{v} &= \frac{v_l \sqrt{\rho_l} + v_r \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \\ \tilde{E} &= \frac{E_l \sqrt{\rho_l} + E_r \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \\ \tilde{B}_x &= \frac{B_{xlsqrt} \rho_l + B_{xR} \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \\ \tilde{B}_y &= \frac{B_{yl} \sqrt{\rho_l} + B_{yR} \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \\ \tilde{B}_z &= \frac{B_{zl} \sqrt{\rho_l} + B_{zR} \sqrt{\rho_r}}{\sqrt{\rho_l} + \sqrt{\rho_r}} \end{aligned} \tag{5.4.39}$$

For the wavespeeds one finally obtains

$$\begin{aligned} s_l &= min\ (q_l - c_{f\,l}, q_{Roe} - c_{f\,Roe}) \\ s_r &= min\ (q_r + c_{f\,r}, q_{Roe} + c_{f\,Roe}) \end{aligned} \tag{5.4.40}$$

- *Formulas for intermediate states*

$$B_x^{HLL} = B_{xl}^* = B_{xr}^* = \frac{S_r B_{xr} - S_l B_{xl}}{S_r - S_l} \tag{5.4.41}$$

$$B_y^{HLL} = B_{yl}^* = B_{yr}^* = \frac{S_r B_{yr} - S_l B_{yl}}{S_r - S_l} \tag{5.4.42}$$

$$B_z^{HLL} = B_{zl}^* = B_{zr}^* = \frac{S_r B_{zr} - S_l B_{zl}}{S_r - S_l} \tag{5.4.43}$$

$$q^* = \frac{\rho_r q_r (S_r - q_r) - \rho_l q_l (S_l - q_l) + P_l - P_r - B_{nl}^2 + B_{nr}^2}{\rho_R (S_r - q_R) - \rho_l (S_l - q_l)} \tag{5.4.44}$$

$$p^* = \rho_K (S_K - q_K)(q^* - q_K) + P_K - B_{nK}^2 + B_n^{*2} \tag{5.4.45}$$

$$\begin{aligned} \rho_K^* &= \rho_K \frac{S_K - q_K}{S_K - q^*} \\ (\rho u)_K^* &= (\rho u)_K \frac{S_K - q_K}{S_K - q^*} + \frac{(P^* - P_K) n_x + B_{nK} B_{xK} - B^* B_x^*}{S_K - q^*} \\ (\rho v)_K^* &= (\rho v)_K \frac{S_K - q_K}{S_K - q^*} + \frac{(P^* - P_K) n_y + B_{nK} B_{yK} - B^* B_y^*}{S_K - q^*} \\ E_K^* &= E_K \frac{S_K - q_K}{S_K - q^*} + \frac{P^* q^* - P_K q_K + B_{nK} (\boldsymbol{B} \cdot \boldsymbol{v})_K - B^* (\boldsymbol{B} \cdot \boldsymbol{v})^*}{S_K - q^*} \end{aligned} \tag{5.4.46}$$

and

$$(\boldsymbol{B} \cdot \boldsymbol{u})^* = \boldsymbol{B}^{HLL} \cdot \boldsymbol{u}^{HLL} \tag{5.4.47}$$

5.4.4 Divergence Free Constraint

5.4.4.1 For Cartesian Grids

The idea of constrained transport is to use simple difference formulas (CD) for the induction equation. To make the scheme second order accurate in time, a time centered approximation is taken for the electric field, so, e.g., for 2D ideal MHD in Cartesian grids:

$$E_z = u B_y - v B_x \tag{5.4.48}$$

and the magnetic field is updated as

$$\begin{aligned} B_{x(i,j)}^{n+1} &= B_{x(i,j)}^{n} + \Delta t \frac{E_{z(i,j+1)} - E_{z(i,j-1)}}{2\Delta y} \\ B_{y(i,j)}^{n+1} &= B_{y(i,j)}^{n} - \Delta t \frac{E_{z(i+1,j)} - E_{z(i-1,j)}}{2\Delta x} \end{aligned} \tag{5.4.49}$$

It is easy to prove that the central difference definition of $\nabla \cdot \boldsymbol{B}$

$$(\nabla \cdot \boldsymbol{B})_{(i,j)} = \frac{B_{xi+1,j} - B_{xi-1,j}}{2\Delta x} + \frac{B_{xi,j+1} - B_{xi,j-1}}{2\Delta y} \tag{5.4.50}$$

is exactly conserved during the time step.

5.4.4.2 For Curvilinear Grids

Introducing the curvilinear magnetic and electric field components for the curvilinear coordinate system(ξ, η, ζ):

$$\begin{aligned} (B_\xi, B_\eta, B_\zeta)^T &= \frac{1}{|J|} J \cdot (B_x, B_y, B_z)^T \\ (E_\xi, E_\eta, E_\zeta)^T &= J^{-1,T} \cdot (E_x, E_y, E_z)^T \end{aligned} \tag{5.4.51}$$

where superscript T indicates the transpose. The Jacobian transformation matrices are

$$J = \begin{pmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{pmatrix}, \quad J^{-1,T} = \begin{pmatrix} x_\xi & y_\xi & z_\xi \\ x_\eta & y_\eta & z_\eta \\ x_\zeta & y_\zeta & z_\zeta \end{pmatrix} \tag{5.4.52}$$

and

$$\begin{aligned} E_x &= vB_z - wB_y \\ E_y &= uB_z - wB_x \\ E_z &= uB_y - vB_x \end{aligned} \tag{5.4.53}$$

The elements of J^{-1} are

$$\begin{aligned} (x_\xi)_{i,j,k} &= \frac{x_{i+1,j,k} - x_{i-1,j,k}}{2\Delta\xi} \\ (x_\eta)_{i,j,k} &= \frac{x_{i,j+1,k} - x_{i,j-1,k}}{2\Delta\eta} \\ (x_\zeta)_{i,j,k} &= \frac{x_{i,j,k+1} - x_{i,j,k-1}}{2\Delta\zeta} \end{aligned} \tag{5.4.54}$$

In the curvilinear variables, the induction equation takes the same form as in the Cartesian case:

$$\begin{aligned} \frac{\partial B_\xi}{\partial t} &= \frac{\partial E_\zeta}{\partial \eta} + \frac{\partial E_\eta}{\partial \zeta} \\ \frac{\partial B_\eta}{\partial t} &= -\frac{\partial E_\zeta}{\partial \xi} + \frac{\partial E_\xi}{\partial \zeta} \\ \frac{\partial B_\zeta}{\partial t} &= -\frac{\partial E_\eta}{\partial \xi} - \frac{\partial E_\xi}{\partial \eta} \end{aligned} \tag{5.4.55}$$

Numerical procedure:

1. calculate curvilinear electric field components(Eq. 5.4.51 and 5.4.53).
2. calculate induction equation according to simple central difference in curvilinear grid (Eq. 5.4.55)
3. update Cartesian field components:

$$\boldsymbol{B}^{n+1}_{x,y,z} = \boldsymbol{B}^{n}_{x,y,z} + \Delta t |J| \frac{J^{-1} \cdot \partial \boldsymbol{B}_{\xi,\eta,\zeta}}{\partial t} \tag{5.4.56}$$

5.5 Boundary conditions for MHD

When electromagnetic waves are incident on a boundary between different media, some of the incident energy crosses the boundary and some is reflected.

In general, fields **E**, **B**, **D**, and **H** will be discontinuous at a boundary between two different media, or at a surface that carries charge or current.

Maxwell's equations in different media in integral form read

$$\oint_S \boldsymbol{D}\cdot d\boldsymbol{S} = Q_{enc} \tag{5.5.1}$$

$$\oint_S \boldsymbol{B}\cdot d\boldsymbol{S} = 0 \tag{5.5.2}$$

$$\oint_C \boldsymbol{E}\cdot d\boldsymbol{l} = -\frac{d}{dt}\int_S \boldsymbol{B}\cdot d\boldsymbol{S} \tag{5.5.3}$$

$$\oint_C \boldsymbol{H}\cdot d\boldsymbol{l} = I_{enc} + \frac{d}{dt}\int_S \boldsymbol{D}\cdot d\boldsymbol{S} \tag{5.5.4}$$

The boundary conditions between two media can be determined using the above formulas.

5.5.1 Transverse Components (normal to the boundary)

Apply Eq. 5.5.1 to a small thin box which extends very slightly into both materials:

1) Volume charge densities do not contribute to Q_{enc} as the box is infinitely thin.
2) For the same reason, the edge of the box does not contribute to the flux.
3) Top and bottom of the box contribute with opposite signs as the two normals have opposite directions.

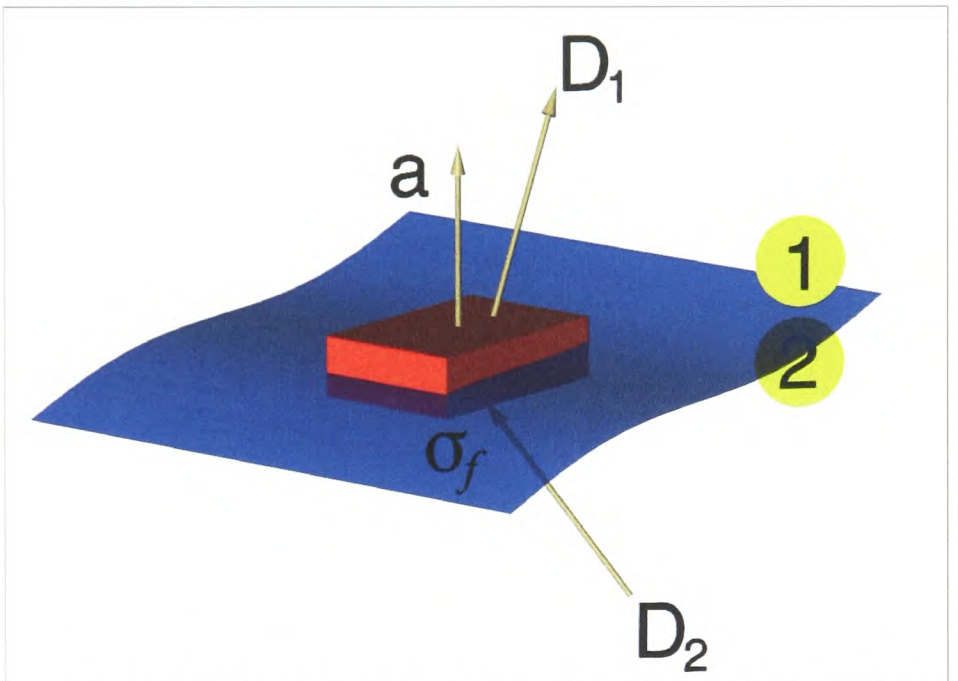


Illustration 5.5.1: shows the transverse components of MHD

So we have

$$\boldsymbol{D}_1 \cdot \boldsymbol{a} - \boldsymbol{D}_2 \cdot \boldsymbol{a} = Q_{enc} = \sigma a \tag{5.5.5}$$

Where $a = |\boldsymbol{a}|$ is the area of the box top, the vector $\boldsymbol{a}$ is directed along its normal, and σ is the surface charge density. Hence for $\boldsymbol{D}^{\perp}$, the normal transverse components of $\boldsymbol{D}$, we have

$$\begin{aligned} \boldsymbol{D}_1^{\perp} - \boldsymbol{D}_2^{\perp} &= \sigma \\ \Rightarrow \epsilon_{r1} \boldsymbol{E}_1^{\perp} - \epsilon_{r2} \boldsymbol{E}_{12}^{\perp} &= \frac{\sigma}{\epsilon_0} \end{aligned} \tag{5.5.6}$$

Where ϵ_{r1} and ϵ_{r2} are respective relative permittivities of the materials. Similarly, starting from Eq. 5.5.2 we have for the transverse components of the magnetic field:

$$B_1^{\perp} - B_2^{\perp} = 0 \tag{5.5.7}$$

5.5.2 Tangential Components (parallel to the boundary)

Apply eq. 5.5.3 to a thin rectangular loop of the length l straddling the boundary:

1) two shorter sides do not contribute to the loop integral, as they are infinitely short;
2) for the same reason, the flux of the magnetic field across the loop also vanishes;
3) top and bottom sides of the loop contribute with opposite signs as they have opposite directions.

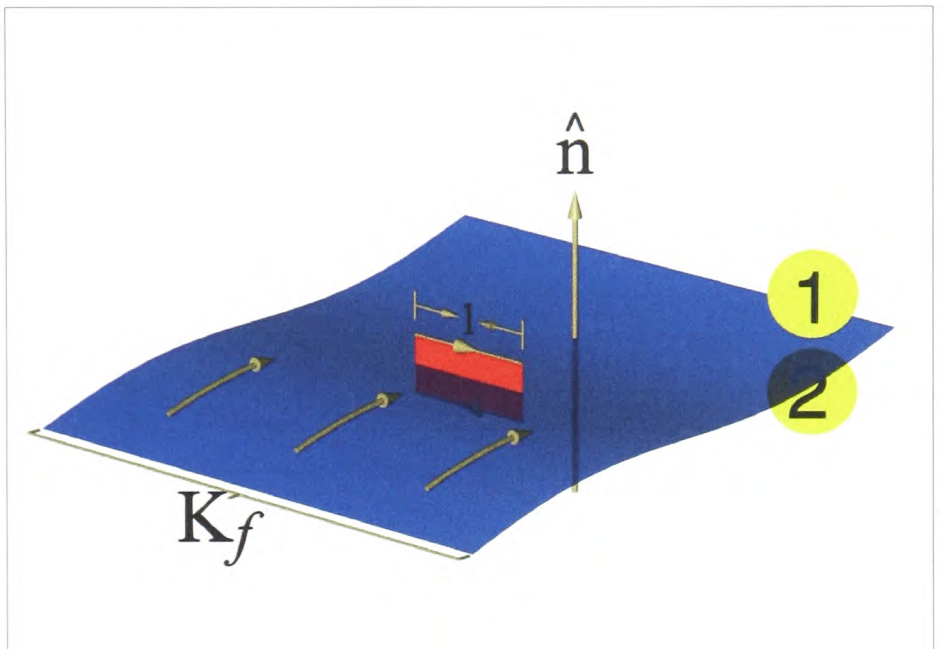


Illustration 5.5.2: shows the transverse components of MHD

So we have

$$\boldsymbol{E}_1 \cdot \boldsymbol{l} - \boldsymbol{E}_2 \cdot \boldsymbol{l} = 0 \tag{5.5.8}$$

Hence, for $E^{\parallel}$ the tangential components of $\boldsymbol{E}$, we have

$$\boldsymbol{E}_1^{\parallel} - \boldsymbol{E}_2^{\parallel} = 0 \tag{5.5.9}$$

Similarly, starting from eq. 5.5.4, we have for the tangential components of the magnetic field

$$\boldsymbol{H}_1 \cdot \boldsymbol{l} - \boldsymbol{H}_2 \cdot \boldsymbol{l} = I_{end} = (\boldsymbol{K} \times n) \cdot \boldsymbol{l} \tag{5.5.10}$$

Where bold K is the surface current density and n is the surface normal. So

$$\begin{aligned} & H_1^{\parallel} - H_2^{\parallel} = \boldsymbol{K} \times n \\ \Rightarrow & \frac{1}{\mu_{r1}} B_1^{\parallel} - \frac{1}{\mu_{r2}} B_2^{\parallel} = \mu_0 \boldsymbol{K} \times n \end{aligned} \tag{5.5.11}$$

Where, μ_{r1} and μ_{r2} are respective relative permeabilities of the two media.

5.5.3 Metallic Boundary Conditions

In a perfect conductor charges are mobile. They move in response to any fields in the fields in the conductor to produce surface charge density σ and surface current density K such that electric and magnetic fields vanish inside the conductor.

So the following previous results, if the medium labelled 2 is a conductor we have

$$\begin{aligned} \boldsymbol{E}_1^{\parallel} &= 0 \\ D_1^{\perp} &= \sigma \\ H_1^{\parallel} &= \boldsymbol{K} \times n \\ B_1^{\perp} &= 0 \end{aligned} \tag{5.5.12}$$

1) In the area just outside a perfect conductor, only normal electric field and only tangential magnetic fields exist.
2) Tangential electrical fields and normal magnetic fields vanish.
3) All fields drops to zero inside a perfect conductor.

These results are utilized for the MHD 2D test case – Riemann Problem 124.

5.6 MHD Divergence Free Numerics

5.6.1 Numerical form of divergence free $\boldsymbol{B}$ field.

In the case of MHD, the induction equation is added, to determine the magnetic inducting field . $\boldsymbol{B}$ This equation is a transport equation, i.e. is time dependent. In addition, $\boldsymbol{B}$ has to satisfy the constraint

$$\nabla \cdot \boldsymbol{B} = 0 \quad \text{at all times n, using Gauss' law,} \quad \oint_{A(U)} \boldsymbol{B} \cdot d\boldsymbol{A} = 0 \tag{5.6.1}$$

5 Multiphysics Equations in JUSTGrid

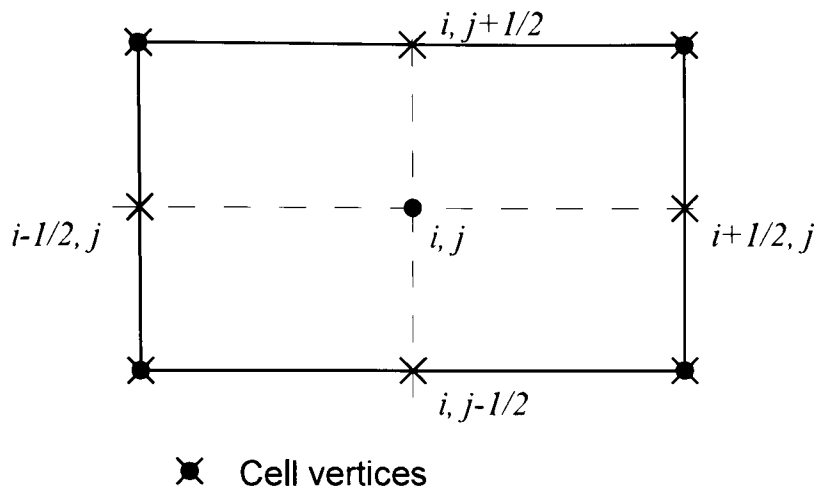


Illustration 5.6.1: 2D Case: finite volume grid variables known only at cell centers. Vector components in the i, j, k directions. (3D) are denoted by indices 1, 2 and 3 respectively. We also can denote components by x, y and z indices, simply considering the Cartesian case.

Let us consider Cartesian coordinates x,y,z and curvilinear coordinates ξ, η, ζ. In physical space (PS) the grid can be irregular, but in computational space (CS) the grid is uniform and orthogonal. Provided, we can determine the normal components of $\boldsymbol{B}$ in the transformed plane and $d\boldsymbol{A}$ is know, then the Cartesian and curvilinear case are the same.

It can be shown that any normal vector (direction) is given by:

$$\hat{n} = D^{-1}(\xi_x, \xi_y, \xi_z) \;:\; \eta - \zeta \; plane$$

$$\hat{n} = D^{-1}(\eta_x, \eta_y, \eta_z) \;:\; \xi - \zeta \; plane$$

$$\hat{n} = D^{-1}(\zeta_x, \zeta_y, \zeta_z) \;:\; \xi - \eta \; plane$$

Note: Since $\boldsymbol{B}$ is a vector it can be expressed in Cartesian coordinates, $\boldsymbol{B} = B_x \hat{e}_x + B_y \hat{e}_y + B_z \hat{e}_z$. The corresponding normal component is simply calculated by $\boldsymbol{B} \cdot \hat{\boldsymbol{n}}$.

In the following Cartesian symbols are used, but it should be clear how to interpret the equations for curvilinear coordinates. Integrating (5.6.1) deliver:

$$B_{x;\, i+1/2} - B_{x;\, i-1/2} + B_{y;\, i+1/2} - B_{y;\, i-1/2} = 0 \qquad (5.6.2)$$

Here, as was said above, $\Delta x = \Delta y = 1$ and only half-integer indices were used.

The induction equation (5.6.3), has to be discretized such that equation (5.6.2) is satisfied at all times, provided the initial solution satisfied $\nabla \cdot \boldsymbol{B} = 0$ numerically.

The induction equation for ideal MHD is:

$$\frac{\partial}{\partial t} \int_V \boldsymbol{B} \cdot dV + \int_V \nabla \times \boldsymbol{E} \; dV = 0 \qquad (5.6.3)$$

It is well known that

$$\boldsymbol{E} = \boldsymbol{v} \times \boldsymbol{B}$$

5.6.2 Divergence free B Field in two dimensions.

First, consider the case $v = (u, v, 0)$; $B = (B_x, B_y, 0)$.

Therefore $E = (0, 0, E_z) = (0, 0, \Omega)$

Calculating $\nabla \times E$ in Cartesian coordinates first:

$$\begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ 0 & 0 & \Omega \end{bmatrix} = \hat{i}\frac{\partial}{\partial y}\Omega - \hat{j}\frac{\partial}{\partial x}\Omega$$

It should be noted that all computations use Cartesian components only. Hence (5.6.3) can be written in 2D:

$$\boxed{\begin{aligned} \frac{\partial}{\partial t}B_x &= \frac{\partial}{\partial y}\Omega = \Omega_y \\ \frac{\partial}{\partial t}B_y &= -\frac{\partial}{\partial x}\Omega = \Omega_x \end{aligned}} \qquad (5.6.4)$$

Discretizing the two equations over a finite volume, one obtains

$$\begin{aligned} B^{n+1}_{x;\,i,j} - B^{n}_{x;\,i,j} &= \Delta t(\Omega^{n}_{i,j+1/2} - \Omega^{n}_{i,j-1/2}) \\ B^{n+1}_{y;\,i,j} - B^{n}_{y;\,i,j} &= -\Delta t(\Omega^{n}_{i+1/2,j} - \Omega^{n}_{i-1/2,j}) \end{aligned} \qquad (5.6.5)$$

Now using equation (5.6.5) at cell faces from equation (5.6.2), we need to calculate:

$$B^{n+1}_{x;\,i+1/2,j} - B^{n}_{x;\,i+1/2,j} = \Delta t(\Omega^{n}_{i+1/2,\,j+1/2} - \Omega^{n}_{i+1/2,\,j-1/2}) \qquad (5.6.6a)$$

$$B^{n+1}_{x;\,i-1/2,j} - B^{n}_{x;\,i-1/2,j} = \Delta t(\Omega^{n}_{i-1/2,\,j+1/2} - \Omega^{n}_{i-1/2,\,j-1/2}) \qquad (5.6.6b)$$

$$B^{n+1}_{y;\,i,j+1/2} - B^{n}_{y;\,i,j+1/2} = \Delta t(\Omega^{n}_{i+1/2,j+1/2} - \Omega^{n}_{i-1/2,j-1/2}) \qquad (5.6.6c)$$

$$B^{n+1}_{y;\,i,j-1/2} - B^{n}_{y;\,i,j-1/2} = \Delta t(\Omega^{n}_{i+1/2,j-1/2} - \Omega^{n}_{i-1/2,j-1/2}) \qquad (5.6.6d)$$

Adding up equations (5.6.6a to 5.6.6d) left hand side of equation (5.6.2) from these four equations (obtained from the discretization of the induction equation) results in:

$$\underbrace{B^{n+1}_{x,\,i+1/2,j} - B^{n+1}_{x;\,i-1/2,j} + B^{n+1}_{y;\,i,j+1/2} - B^{n+1}_{y;\,i,j-1/2} = 0}_{\nabla \cdot B \text{ at } n+1 \text{ timestep}}$$

This can be checked from the above equations, but there is also a geometrical method. It was assumed that field was divergence free at time level n.

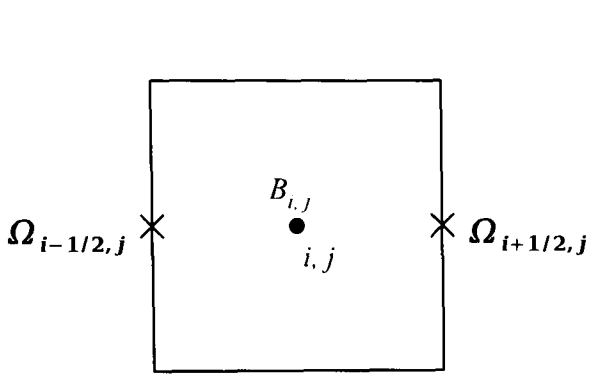


Illustration 5.6.3: Discretization of induction equation

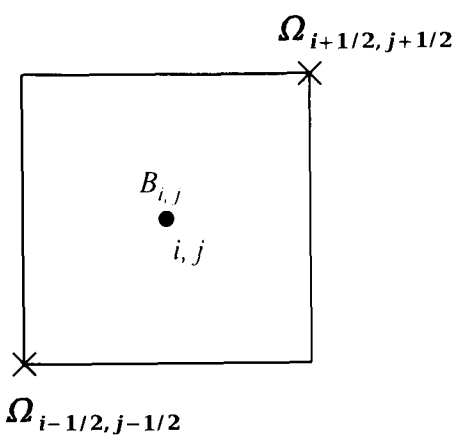


Illustration 5.6.2: Discretization of $\nabla\cdot\vec{B}$ in 2D case.

Result: Utilizing the induction equation in 2D in form of equation (5.6.4) with the finite volume discretization (5.6.5) automatically satisfies the numerical version of $\nabla\cdot\boldsymbol{B} = 0$ as given by equation (5.6.2).

The rotation has the same form in the curvilinear coordinates, since the Cartesian rotation was used in in equation (5.6.4), that is:

$$\frac{\partial B^1}{\partial t} = -\frac{\partial E^3}{\partial \eta} + \frac{\partial E^2}{\partial \zeta} \; ; \qquad \frac{\partial B^2}{\partial t} = \frac{\partial E^3}{\partial \xi} - \underbrace{\frac{\partial E^1}{\partial \zeta}}_{=0} \; ; \qquad \underbrace{\frac{\partial B^3}{\partial t}}_{=0} = \frac{\partial E}{\partial \eta}$$

5.6.3 Divergence free B field in three dimensions.

Now we consider the 3D case. we have:

$$\boldsymbol{E} = (E_x,\ E_y,\ E_z)$$

$$\nabla \times \boldsymbol{E} = \begin{bmatrix} \hat{i} & \hat{i} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ E_x & E_y & E_z \end{bmatrix} = \hat{i}\left(\frac{\partial}{\partial y}E_z - \frac{\partial}{\partial z}E_y\right) - \hat{j}\left(\frac{\partial}{\partial x}E_z - \frac{\partial}{\partial z}E_x\right) + \hat{k}\left(\frac{\partial}{\partial x}E_y - \frac{\partial}{\partial y}E_x\right)$$

For the sake of simplicity we use $(\Delta,\ \Lambda,\ \Omega)$

$$(E_x,\ E_y,\ E_z) = (C,\ D,\ E)$$

$$\nabla \times \boldsymbol{E} = \hat{i}(-D_y + E_z) - \hat{j}(E_x - C_z) + \hat{k}(D_x - C_y)$$

Rotation equation (induction equation)

$$B_\xi = B_x\xi_x + B_y\xi_y + B_z\xi_z$$

$$M_x = \begin{bmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{bmatrix}$$

$$M_\xi M_x = I$$

$$\begin{bmatrix} B^1 \\ B^2 \\ B^3 \end{bmatrix} = M_x \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} \qquad M_\xi = \begin{bmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{bmatrix} \qquad \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} = M_\xi \begin{bmatrix} B^1 \\ B^2 \\ B^3 \end{bmatrix}$$

$${}_1B^{n+1}_{i,j} - {}_1B^n_{i,j} = \Delta t(\Omega^n_{i,j+1/2} - \Omega^n_{i,j-1/2})$$

$${}_2B^{n+1}_{i,j} - {}_2B^n_{i,j} = \Delta t(\Omega^n_{i+1/2,j} - \Omega^n_{i-1/2,j})$$

$${}_3B^{n+1}_{i,j} - {}_3B^n_{i,j} = \Delta t(\Omega^n_{i+1/2,j+1/2} - \Omega^n_{i-1/2,j-1/2})$$

5.6.4 Equivalence of curvilinear grid in physical space and Cartesian grid in computational space

Note: Only Cartesian grids need to be considered for $\nabla \cdot \boldsymbol{B} = 0$. We have shown that the transformed equation have the general form:

$$\frac{\partial}{\partial t}\int_V \hat{\boldsymbol{U}}\, d\xi\, d\eta\, d\zeta + \int_A \hat{\boldsymbol{F}}\, d\eta\, d\zeta + \int \hat{\boldsymbol{G}}\; d\xi\, d\zeta + \int \hat{\boldsymbol{H}}\; d\xi\, d\eta = \int_V \hat{\boldsymbol{W}}\, d\xi\, d\eta\, d\zeta$$

where $\hat{\boldsymbol{F}}$, $\hat{\boldsymbol{G}}$, and $\hat{\boldsymbol{H}}$ are flux vectors that are orthogonal to their respective faces. In the computational space coordinate directions denoted by indices i, j, and k are orthogonal. Each grid is uniform. Any grid in physical space is equivalent to a Cartesian grid with uniform grid spacing in computational space. Therefore, in the following, only the integral form of the divergence free magnetic inductions field in Cartesian space considered.

Note: In the induction equation we have the term: $(\boldsymbol{vB} - \boldsymbol{Bv})$

In 2D we have

$$\frac{\partial}{\partial t}B_x + \underbrace{(uB_x - B_x u)}_{=0} + (uB_y - B_x v) + \underbrace{(uB_z - B_x w)}_{=0 \text{ in 2D}} = 0$$

$$\frac{\partial}{\partial t}B_y + (vB_x - B_y u) + \underbrace{(vB_y - B_y v)}_{=0} + \underbrace{(vB_z - B_y w)}_{=0 \text{ in 2D}} = 0$$

$$\frac{\partial}{\partial t}B_z + \underbrace{(wB_x - B_z u)}_{=0 \text{ in 2D}} + \underbrace{(wB_y - B_z v)}_{=0 \text{ in 2D}} + \underbrace{(wB_z - B_z w)}_{=0} = 0$$

This leads to exactly the same equations as in (5.6.4) although we have the divergence form of the

$\frac{\partial}{\partial t}\int \boldsymbol{B}\, dV + \int (\boldsymbol{vB} - \boldsymbol{Bv}) \cdot d\boldsymbol{A} = 0$ induction equation.

6 Computational and physics model Validation in *JUSTGRID*

6.1 "Write once run anywhere"

The compiled Java classes (binaries) of the ***JUSTGRID*** framework and the GRX Tools where successfully tested on the following computer system:

JVM Version(s)	Computer Model	Processor Architecture	Operating System
1.4.x, 1.5.x, 1.6.0	Sun Microsystems, Sun Fire V880, 8 CPUs, 32GB Memory	SPARC	Solaris 9 Solaris 10
1.4.x	Sun Microsystems, Enterprise 10000, 64 CPU, 192GB Memory	SPARC	Solaris 9
1.4.x, 1.5.x, 1.6.0	Sun Microsystems, Ultra 40, 2 CPU, 8GB Memory	AMD Dual Core Opteron	Solaris 10, Windows XP 32 Bit, Linux Ubuntu 6.06
1.4.x, 1.5.x, 1.6.0	Dell Latitude D820, 1 CPU, 2GB Memory	Intel Core Duo	Linux Ubuntu 6.06 Windows XP 32 Bit
1.5.0	Apple MacBook Pro, 1 CPU, 2GB Memory	Intel Core Duo	MacOS X 10.4.8
1.5.0	Apple PowerBook G4, 1CPU, 768MB Memory	Power PC	MacOS X 10.4
1.6.0	PC, 1 CPU, 1GB Memory	Intel Pentium 4	Windows Vista RC2
1.4.x, 1.5.x, 1.6.0	PC, 1 CPU, 1GB Memory	Intel Pentium 4	Linux Mandriva 2007 Linux Fedora Core 5

Table 6.1.1: Computer systems successfully tested with JUSTGrid.

6.2 Loaders and Writers

A good way to validate a loader is to write a writer to be used in parallel. First, load a data structure into JUSTGrid and if no Java Exception will be thrown write the structure just loaded into a different file. This just written data file will be named stage 1 data file. A stage 1 data file need not be exactly the same as the original data file loaded in the first step. There may be differences with space characters or rounded errors for double number . Therefore it is difficult to compare a stage 1 data file with an original data file. The next step is to load the stage 1 data file and to write it out again in a different file (stage 2 data file). Now stage 1 and stage 2 data file must exactly be the same and are supposed not to change in any way even if one writes a stage 3 data file etc.

6.3 Topology handling for complex geometries

The grid topology is the information about the connectivity between neighbouring blocks, the orientation of the matching faces, and the physical boundary conditions.

6.3.1 Connectivity

JUSTGRID recognises the connectivity of a block automatically. For validation purposes several large grids were read in, and the connectivity information was written out and compared with the known connectivity.

6.3.2 Orientation

JUSTGRID recognises the orientation of the neighbouring faces automatically. To validate the recognised orientation several large grids were read in, and the orientation information was written out and compared with the known orientation information.

6.4 Boundary Data Exchange

A Java program was built to test the boundary exchange for all 8 possible orientations. The picture below was produced by this program and shows (utilizing the Java 3D API) a 9 block grid with cell midpoints.

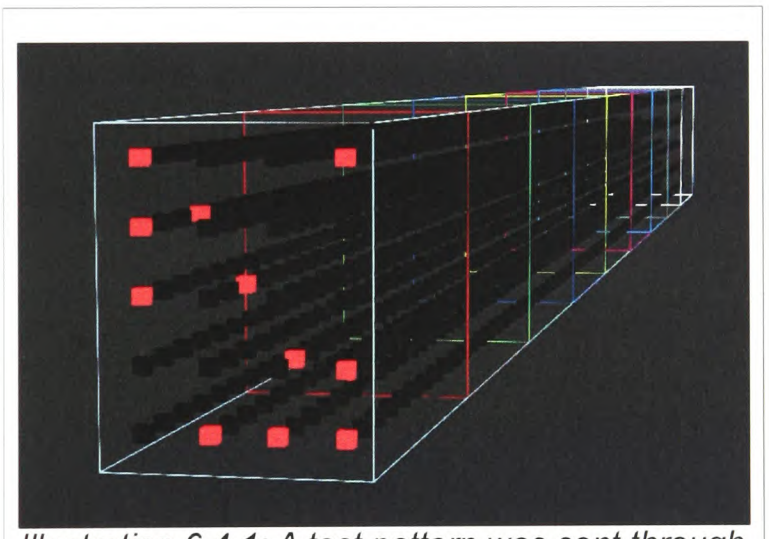


Illustration 6.4.1: A test pattern was sent through the solution domain step by step.

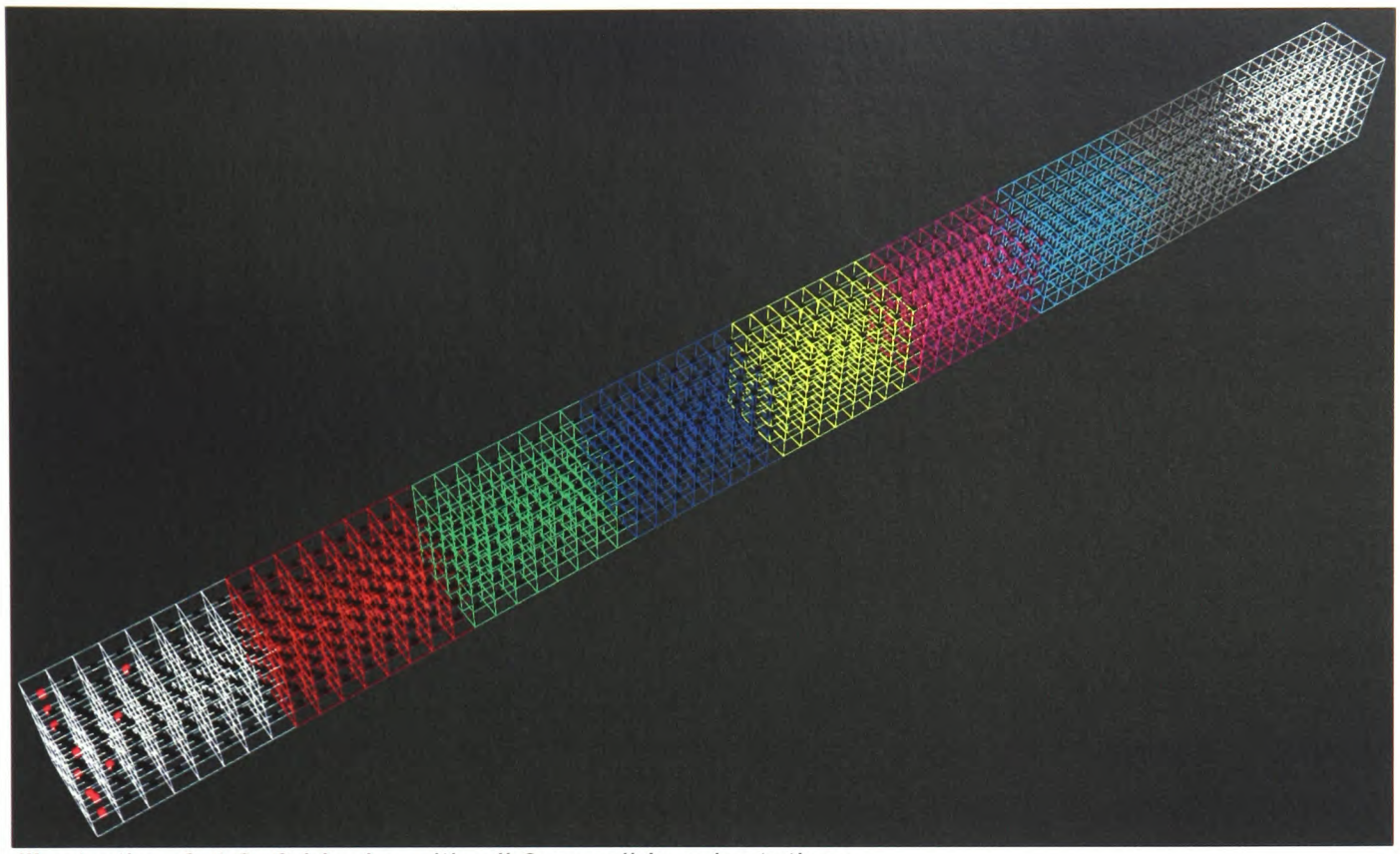


Illustration 6.4.2: 9 blocks with all 8 possible orientations

The test solution domain consists of 9 blocks with 5 x 4 x 8 cells in each block.

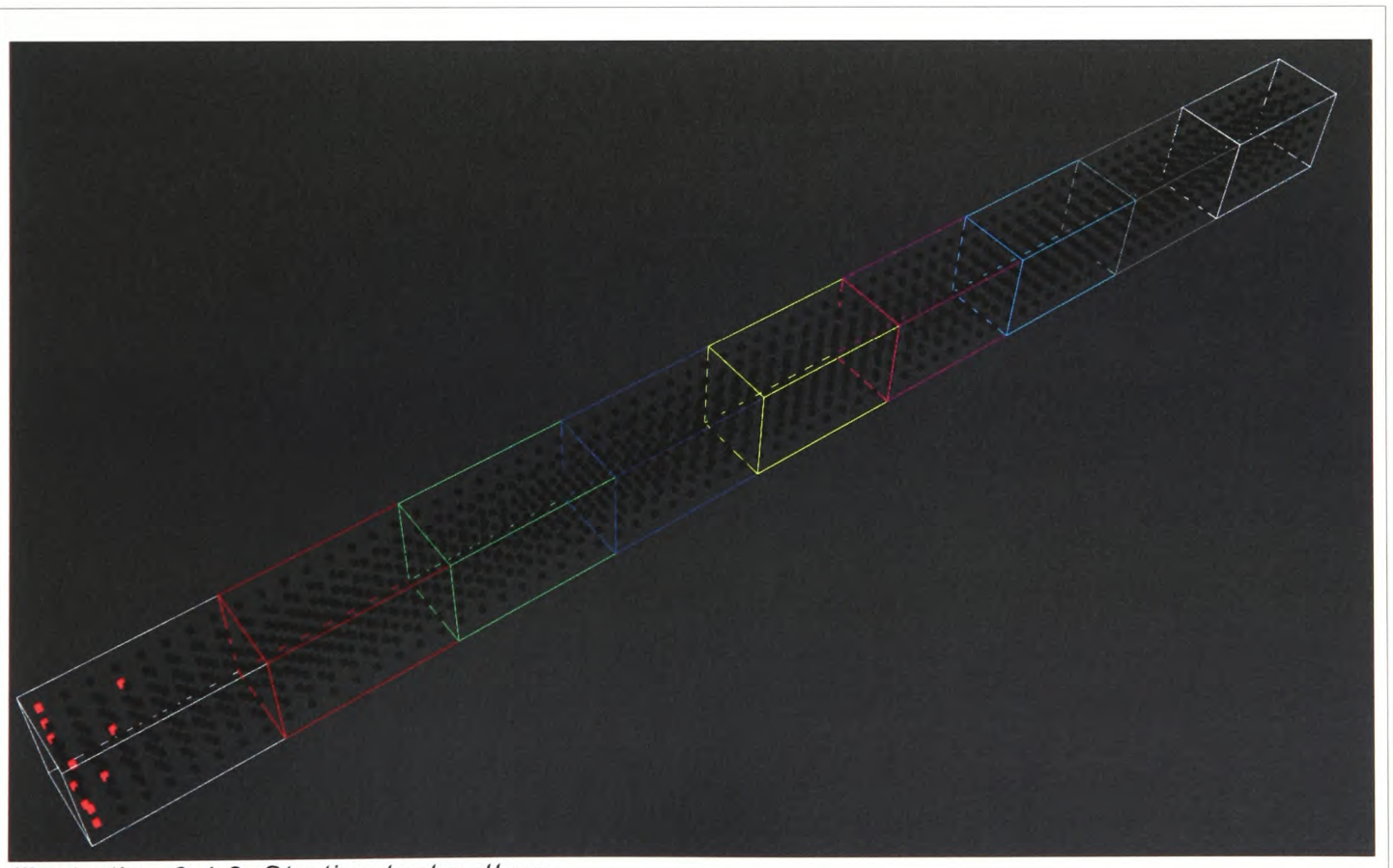


Illustration 6.4.3: Starting test pattern

During the observation of the test pattern running through the solution domain the cell boundaries are not shown for better view of the cell midpoints.

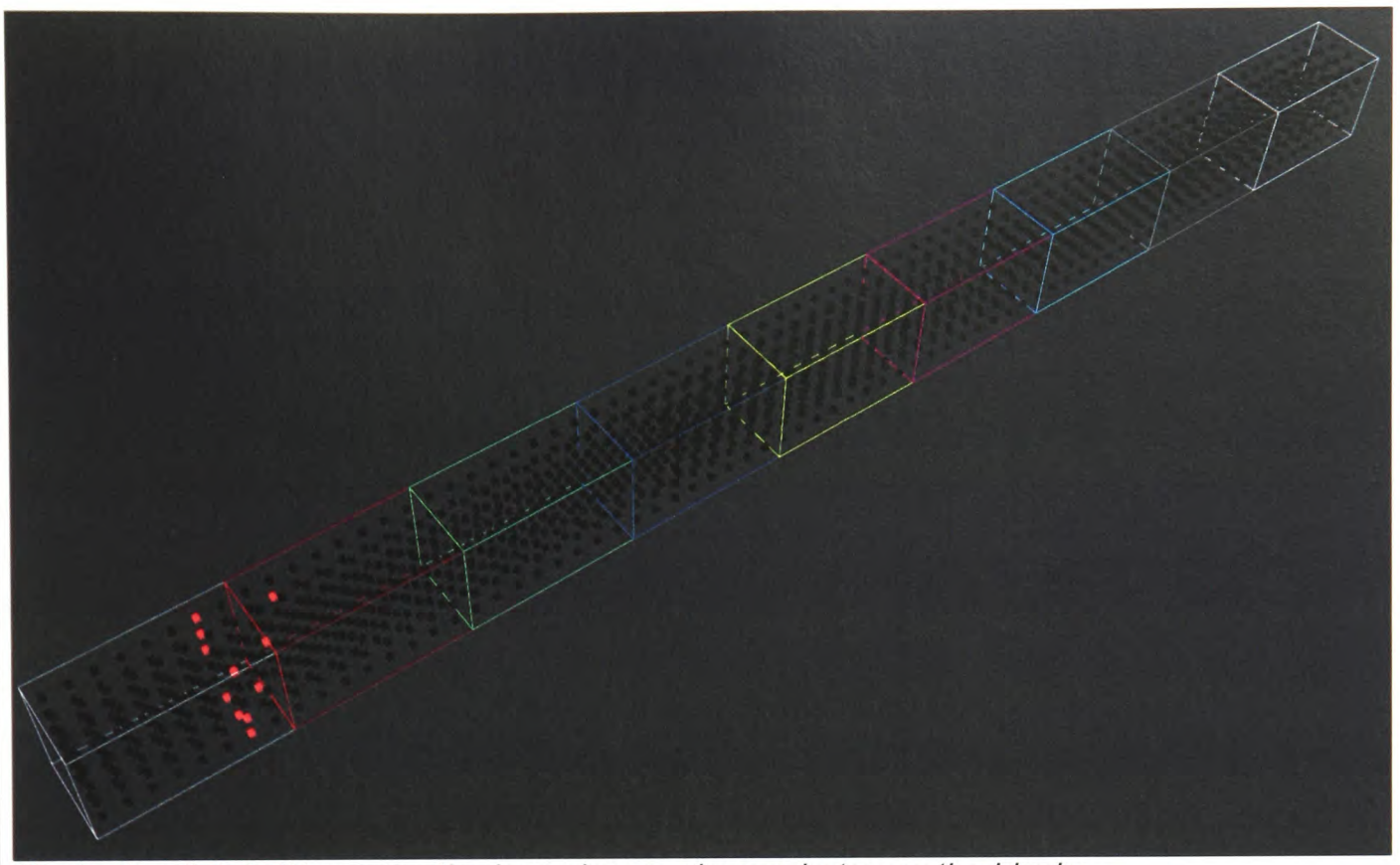


Illustration 6.4.4: Observing the boundary exchange between the blocks.

Every singe step (iteration) is done by a manual mouse-click to carefully inspect the boundary exchange on the block boundaries.

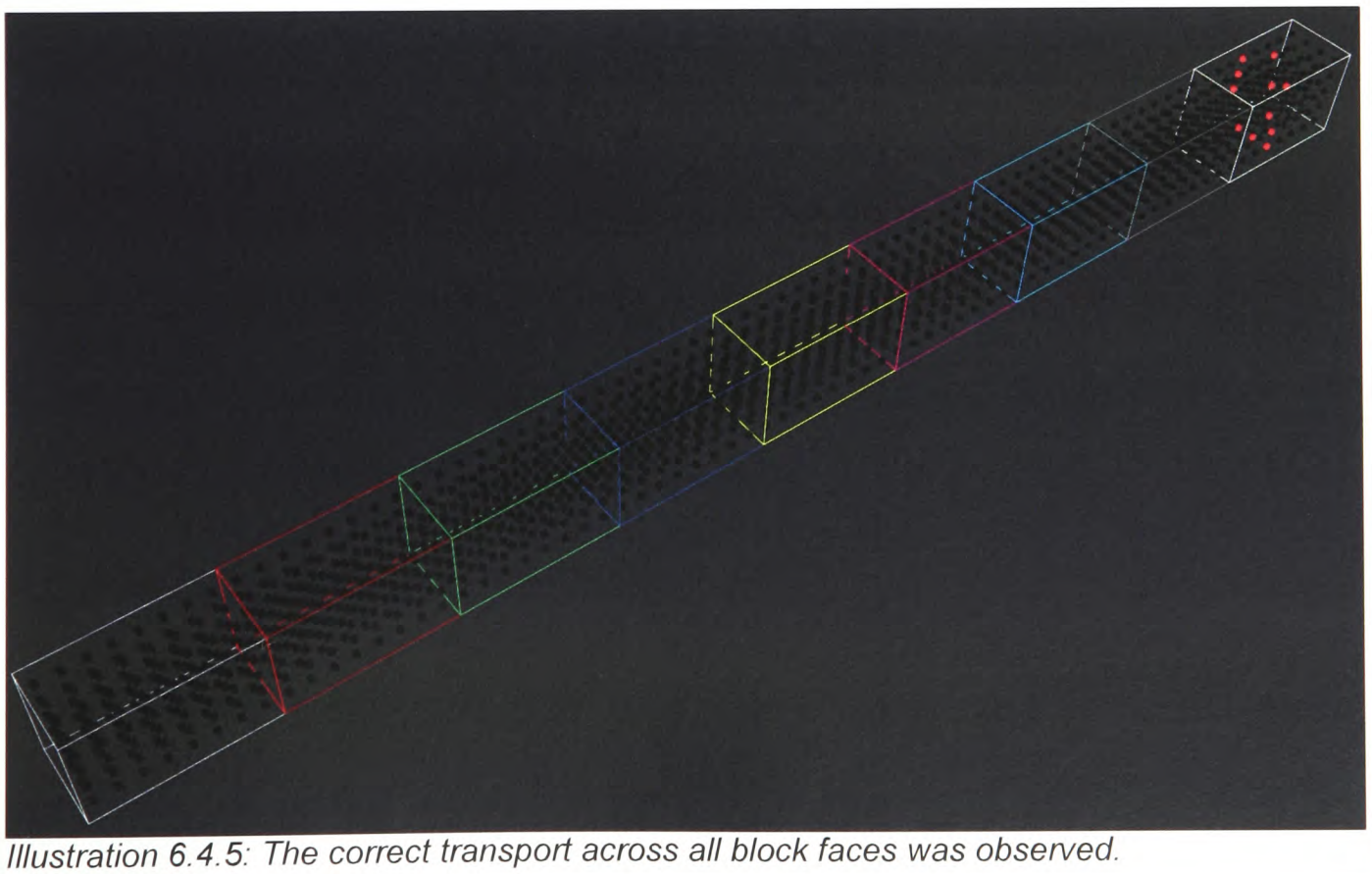


Illustration 6.4.5: The correct transport across all block faces was observed.

After the successful transport of the test pattern through the whole solution domain the test was also successfully done in th e reverse direction, from the way back to the start

6.5 Numerics

Several simulations were performed to ensure the correct working of the different layers of the ***JUSTGrid*** framework. ***JUSTGrid*** is the core of ***JUST,*** the Java Ultra simulator technology. Hence utmost care was taken to prove that ***JUSTGrid*** works absolutely correct. The solvers implemented in package ***JUSTSolver*** will test the ***JUSTGrid*** functionality, performance and efficiency. At present , numerical and physical accuracy of the computational scheme and physical validity of the model, however, are of lesser importance. Therefore, in some computations, a Laplace solver was used to mimic a CFD problem, see below.

6.5.1 1 Block - JUSTSolver - Laplace 3D

The Laplace solver uses Dirichlet boundary conditionsthat means, in this case inflow (v=1) and wall (v=0) boundaries have fixed values. At the outflow boundary extrapolation is used that means, values will be transported out of the solution domain.

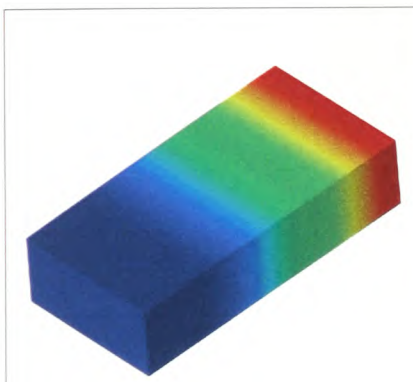


Table 6.5.1: Monoblock result for a Laplace 3D computation.

This is a real simple mono block test case to validate the Laplace solver numerics. After a few iterations an equilibrium is achieved between the inflow (1=red) and the wall (0=blue) faces.

6.5.2 7 Blocks - JUSTSolver - Laplace 3D

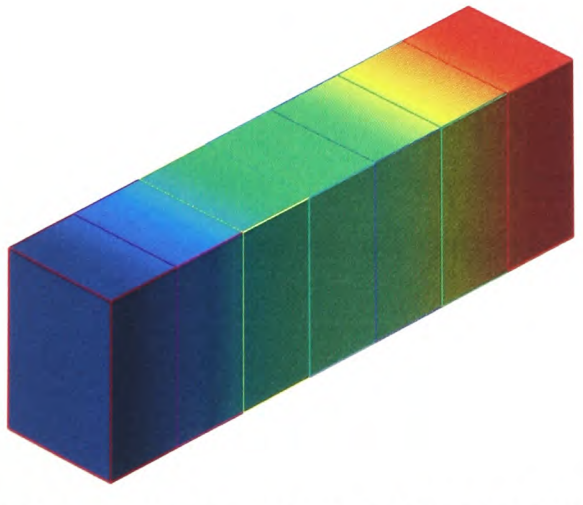


Table 6.5.2: Laplace 3D result for a simple 7 block rectangular grid.

With this sample the simplest communication between the block boundaries is validated. In this case the blocks are not rotated against their neighbours. It should be noted, in order to provide complete geometrical fexibility, each block needs to have its own local coordinate system. Therefore, the correct transformation of information across block faces has to be ensured.

6.5.3 Bump

The next example is a 22 block grid for the well known aerodynamic example from ONERA, called the ONERA bump. This grid has 16,038 points and the solution domain uses 11,264 cells, without halo cells.

6.5.3.1 JUSTSolver - Laplace 3D

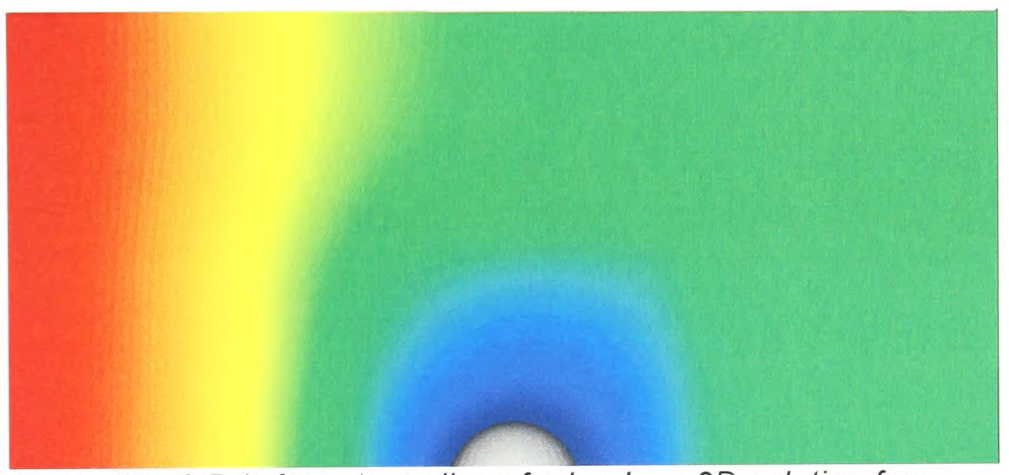


Illustration 6.5.1: A contour slice of a Laplace 3D solution for a bump.

At first the bump was tested with the ***JUSTSolver*** Laplace 3D.

6.5.3.2 Euler 3D

For the first numerics test for a real CFD problem using the ***JUSTSolver*** Euler 3D, supersonic Mach 2.0 free stream conditions are used.

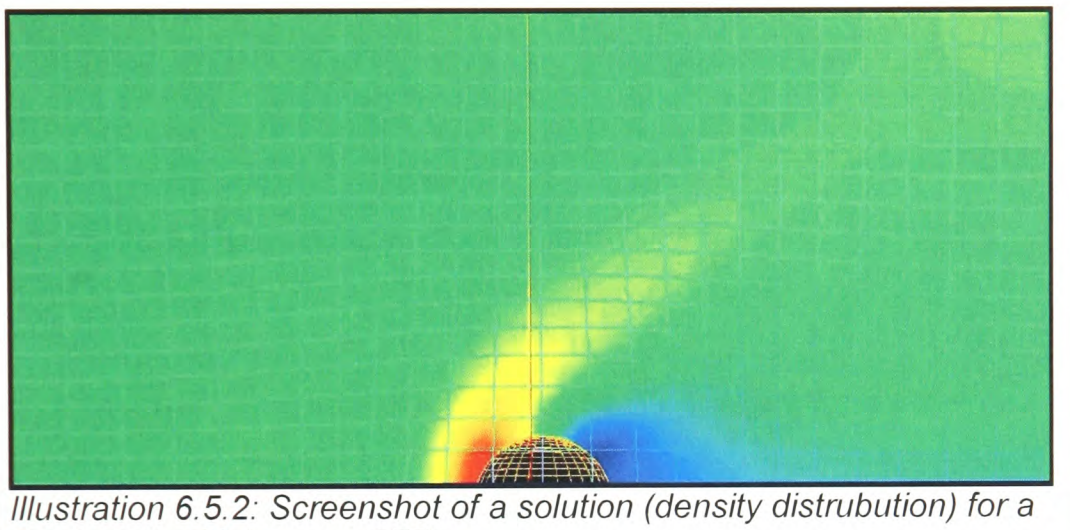


Illustration 6.5.2: Screenshot of a solution (density distrubution) for a bump using Metacomp CFD++

The illustration above shows a solution from an unstructured grid of the bump using the commercial flow solver Metacomp CFD++ as a reference. The CFD++ solution is 2nd order accurate. All simulations with ***JUSTSolver*** Euler 3D are 1st order accurate only.

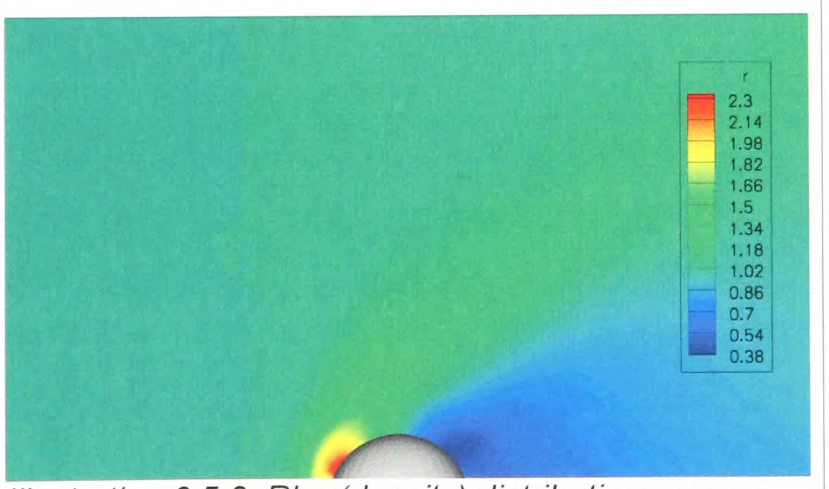


Illustration 6.5.3: Rho (density) distribution over a bump after 1000 iteration with JUSTSolver Euler 3D

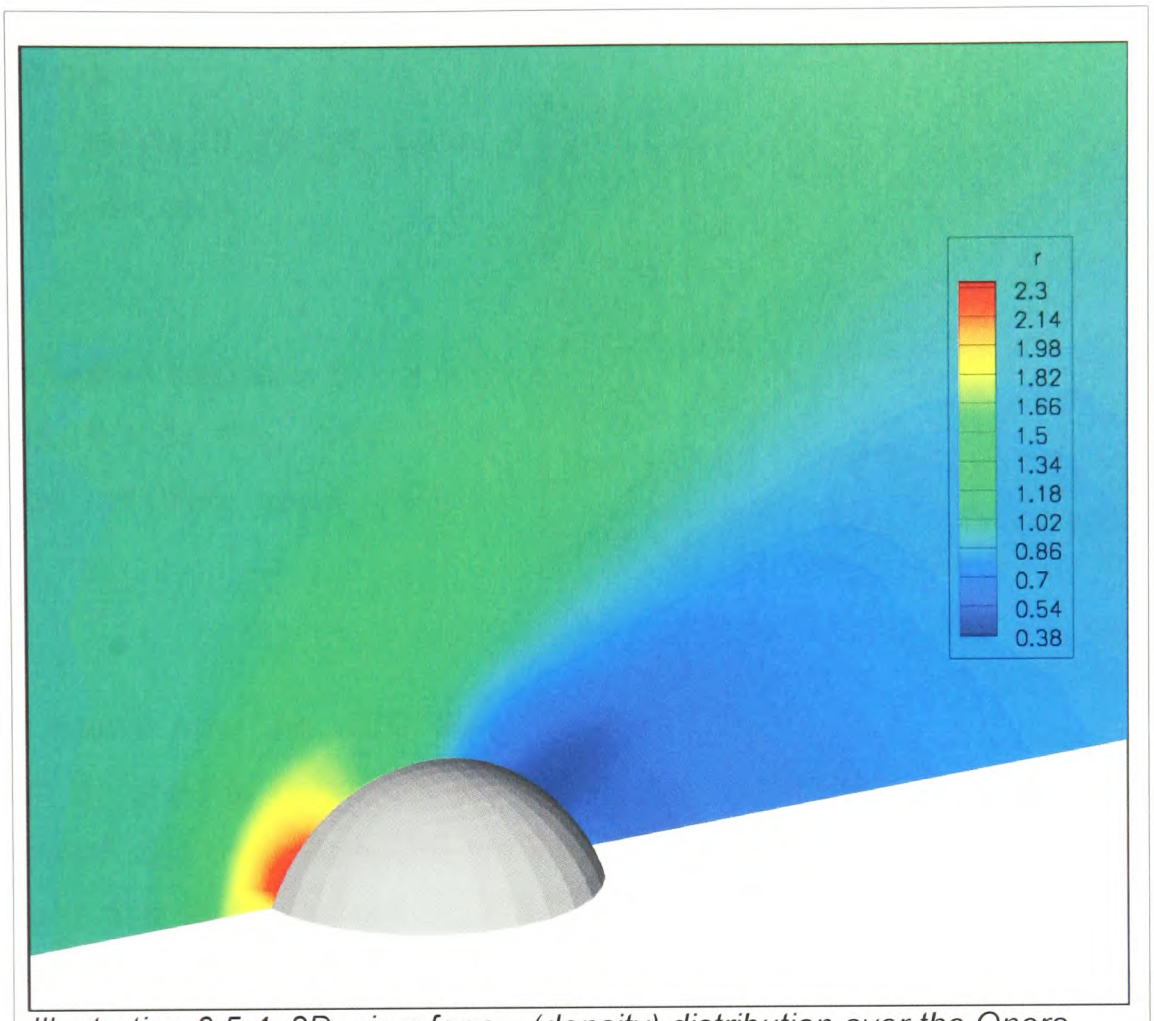


Illustration 6.5.4: 3D view for a ρ (density) distribution over the Onera bump using JUSTSolver Euler 3D

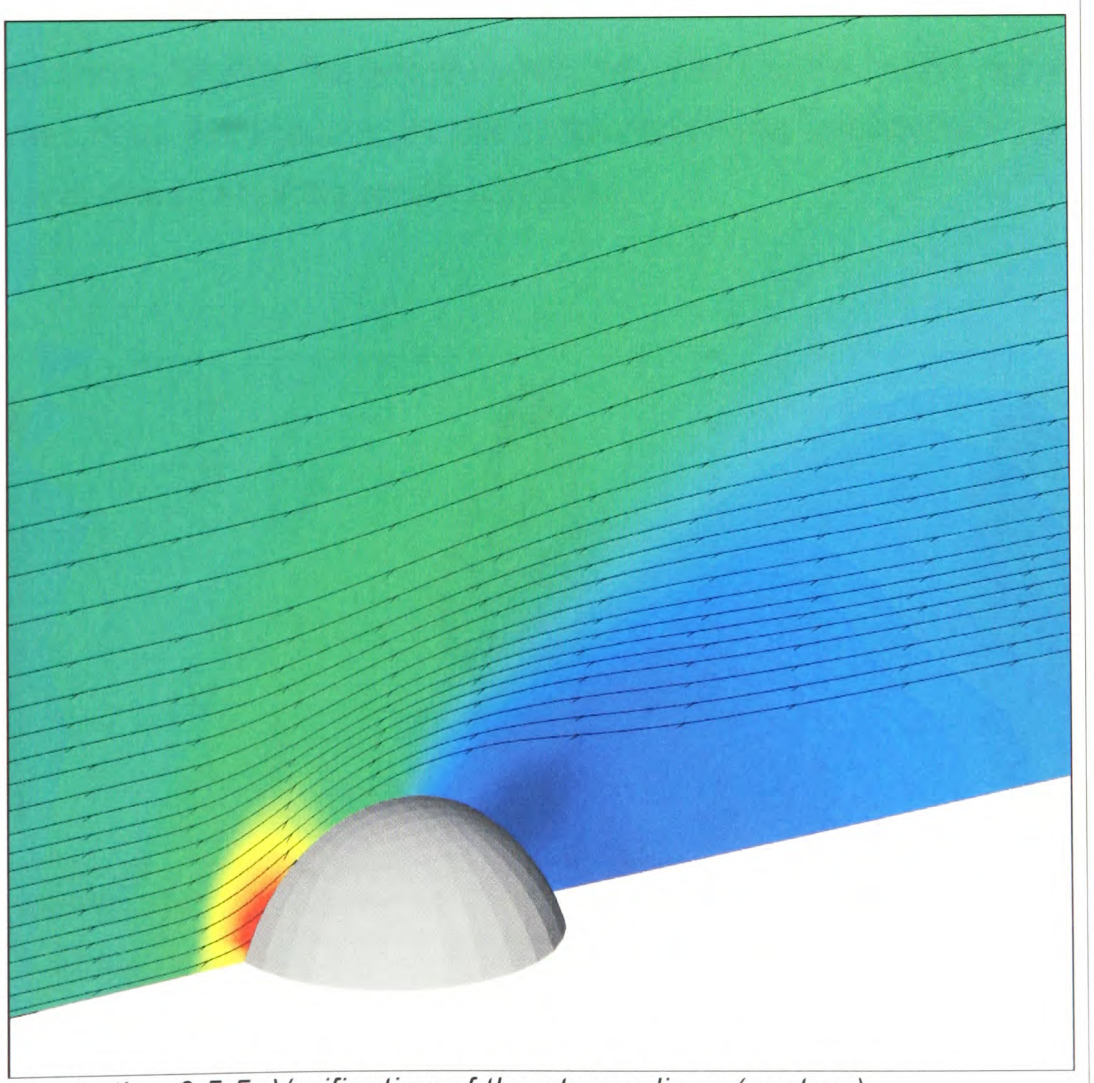


Illustration 6.5.5: Verification of the stream lines (vectors)

6.5.4 3D Cone

Simple 3D cone, 8 blocks, 5,832 grid points, 4,096 cells without halo cells.

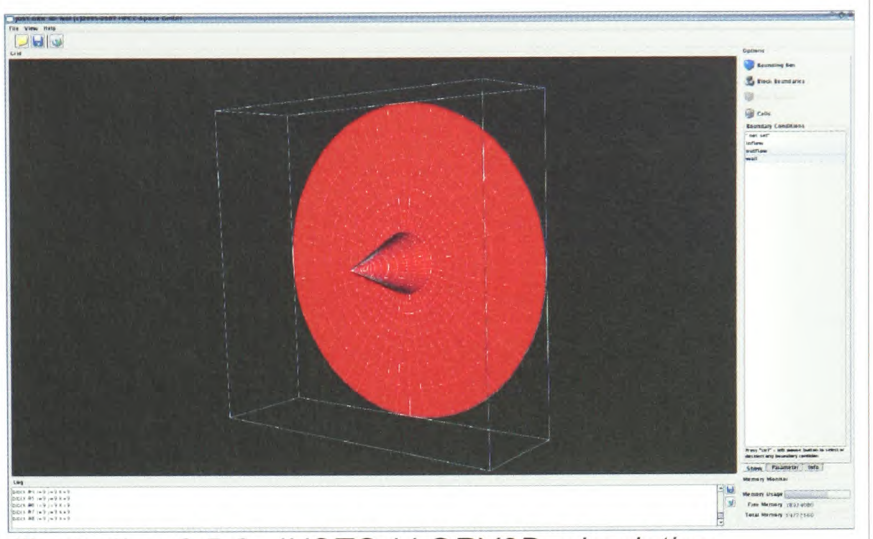


Illustration 6.5.6: JUSTGrid GRX3D, simulation preparation tool, showing the grid of the cone wall and the outflow face.

The cone was selected because it is a well known test case. It was thus possible to nearly check the complete functionality of ***JUSTGRID***.

This Illustration shows how the ***JUSTGRID*** GRX3D Tool can be used to prepare a simulation run. ***JUSTGRID*** GRX3D is also based on the ***JUSTGRID*** framework and uses the same loaders and utility classes as ***JUSTSOLVER*** to visualize a grid. It is the very first test to check if ***JUSTGRID*** can handle a given grid. In addition to the visualization one can specify solver specific parameters like „max number of iterations", „Mach number" or „Dt". These parameters are **not** predefined but depend on the selected solver.

6.5.4.1 JUSTSOLVER Laplace 3D

The Laplace solver uses Dirichlet boundary conditions, that means in this case inflow (v=1) and wall (v=0) boundaries have fixed values. At the outflow boundary extrapolation is used that means the value will be transported out of the solution domain.

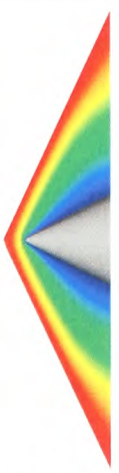


Illustration 6.5.7: JUSTSolver Laplace 3D, Cone, showing one slice on the y-plane.

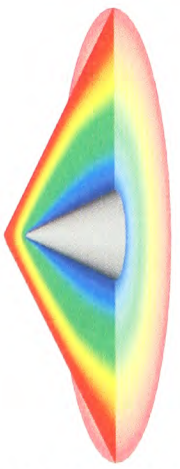


Illustration 6.5.8: JUSTSolver Laplace 3D, Cone, showing the outflow boundary.

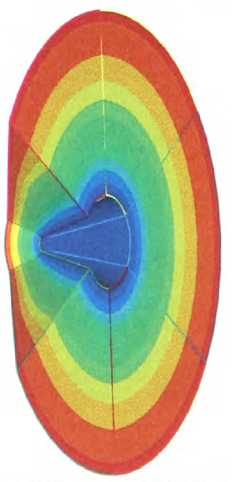


Illustration 6.5.9: JUSTSolver Laplace 3D, Cone, showing block edges with one deactivated block.

The following tasks could be verified with this simple test case:

1. Parallelization - ***JUSTGrid*** starts one Thread per block and one monitor thread. All available processors are been used by the simulation. If the computational grid has less blocks than the compute system's number of processors then the surplus processors will be idle.
2. Synchronization - ***JUSTGrid*** implements a loose synchronization between the neighbouring blocks. Therefore it is possible that neighbour blocks are one iteration ahead.
3. Communication - ***JUSTGrid*** is also responsible for the boundary update between the neighbouring blocks. One can specify any number of halo cells.

6.5.4.2 JUSTSolver Euler3D (1st order, explicit, structured multiblock) compared with CFD++ (2nd order, unstructured)

To test the numerical correctness of ***JUSTSolver*** Euler3D the result of the cone simulation is compared with the result from a commercial CFD solver (CFD++).

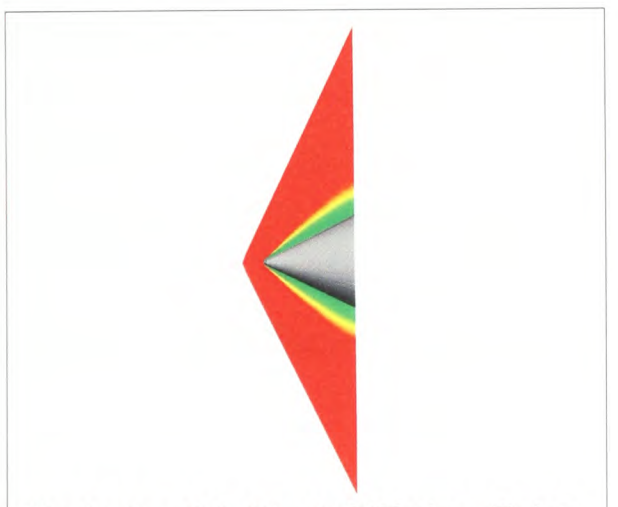


Illustration 6.5.10: JUSTSolver Euler 3D showing Mach number solution in symmetry plane.

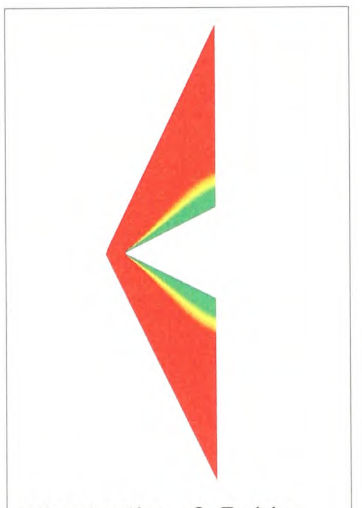


Illustration 6.5.11: CFD++ comparison simulation.

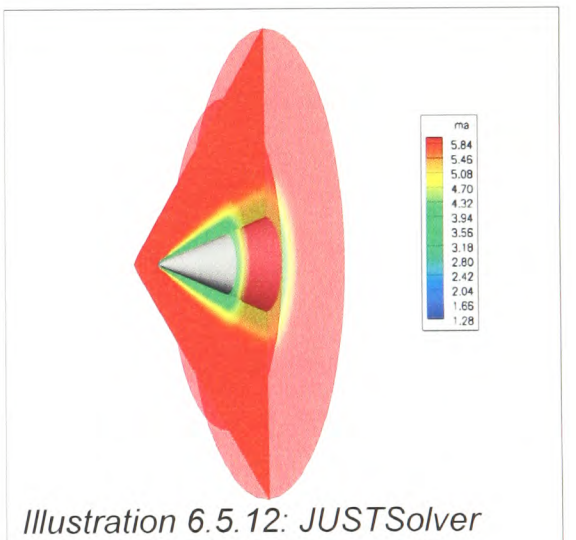


Illustration 6.5.12: JUSTSolver Euler 3D with legend and two slices.

Showing the Mach number distribution for a Mach 6.0 simulation after 2,000 iterations. Although the ***JUSTSolver*** Euler3D simulation is only 1st order accurate, compared with the CFD++ result the Mach number distribution differs not much.

6.5.5 European Experimental Test Vehicle (EXTV)

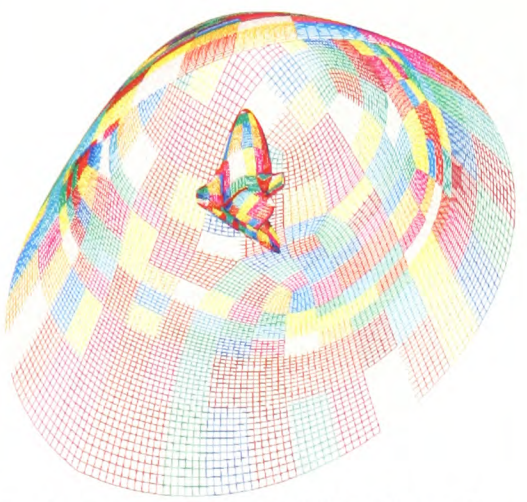


Illustration 6.5.13: GridPro™ grid, showing inflow and wall boundaries.

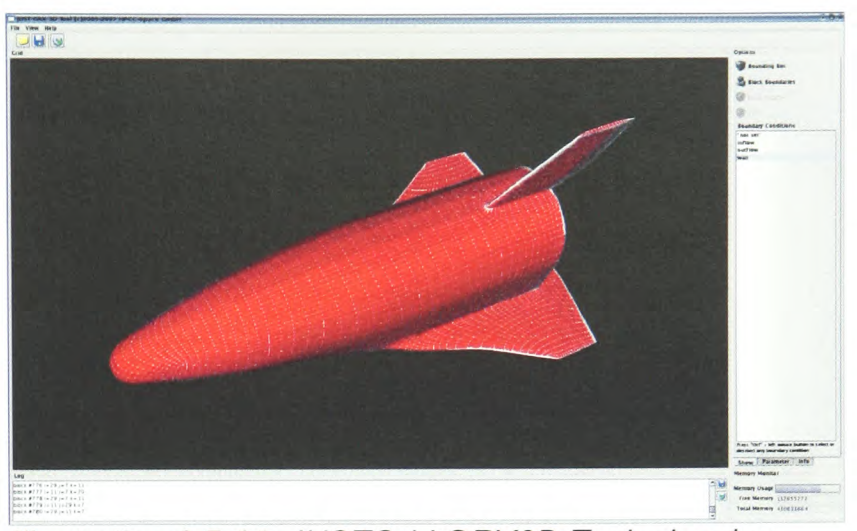


Illustration 6.5.14: JUSTGrid GRX3D Tool, showing EXTV wall boundary.

With 780 blocks, 755,300 grid points and 538,752 cells without halo cells, the EXTV grid is a serious test case for larger simulations. The size of this test case is ideal for efficiency and speedup tests, because it has a reasonable number of blocks and produces enough numerical work load to achieve a homogeneous dynamic load balancing across all available processors. The simulations were run on a Sun Microsystems *Sun Fire V880* server with 8 UltraSPARC III processors (1.2GHz) and 32GByte main memory running Solaris 10 06/06.

6.5.5.1 JUSTSolver Laplace 3D

6 Computational and physics model Validation in JUSTGrid

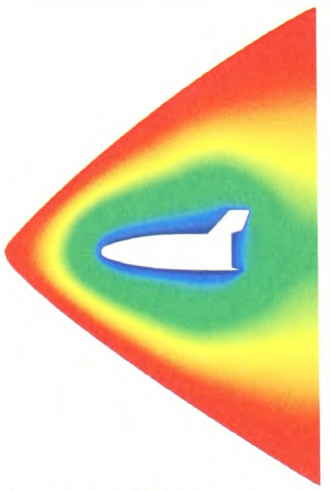


Illustration 6.5.15: JUSTSolver Laplace 3D, EXTV showing one slice at the y-plane.

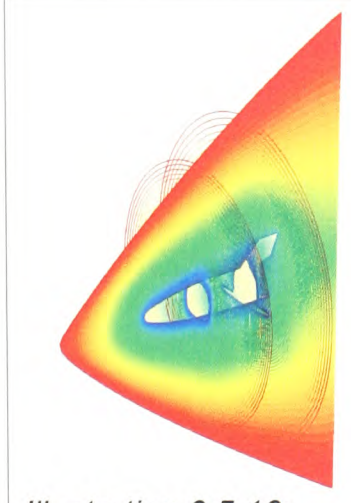


Illustration 6.5.16: JUSTSolver Laplace 3D, EXTV 3D view.

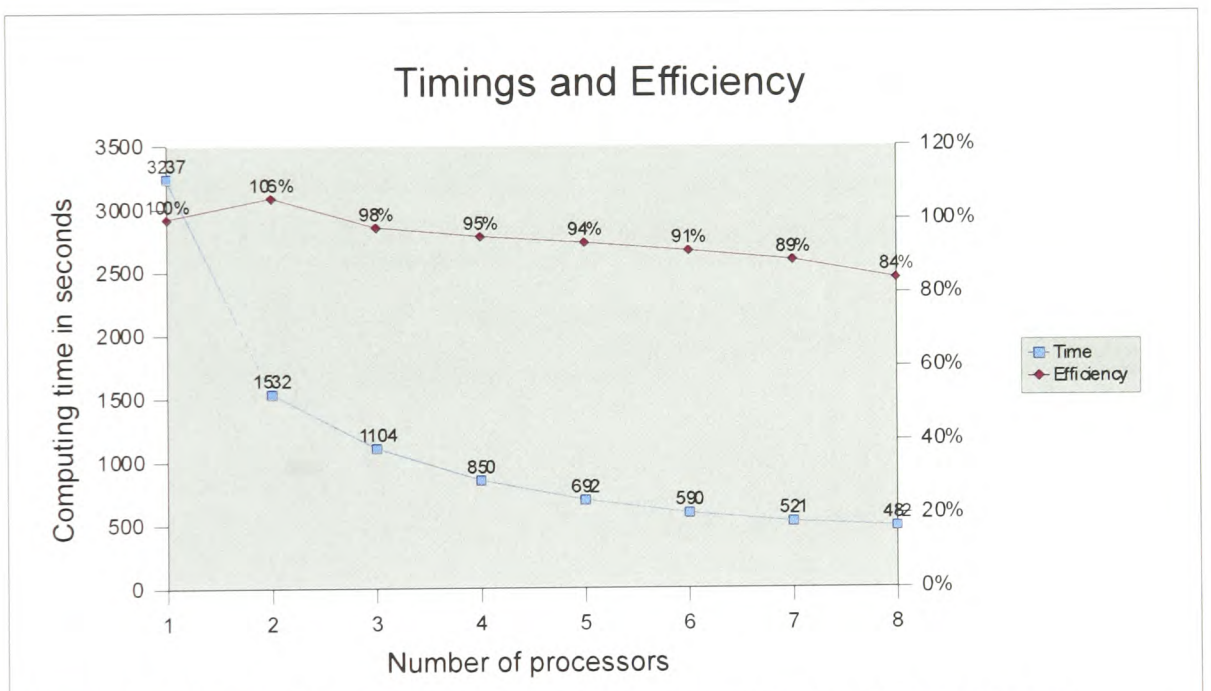


Illustration 6.5.17: Timings and efficiency results for 1 to 8 processors, running 2,000 iterations with JUSTSolver Laplace 3D on a 780 blocks EXTV grid.

The ***JUSTSolver*** Laplace 3D is again used to test parallelization, synchronization and communication features of ***JUSTGrid*** for this large configuration. In order to produce sufficient numerical load the computation was done for 2,000 iterations.

6 Computational and physics model Validation in JUSTGrid

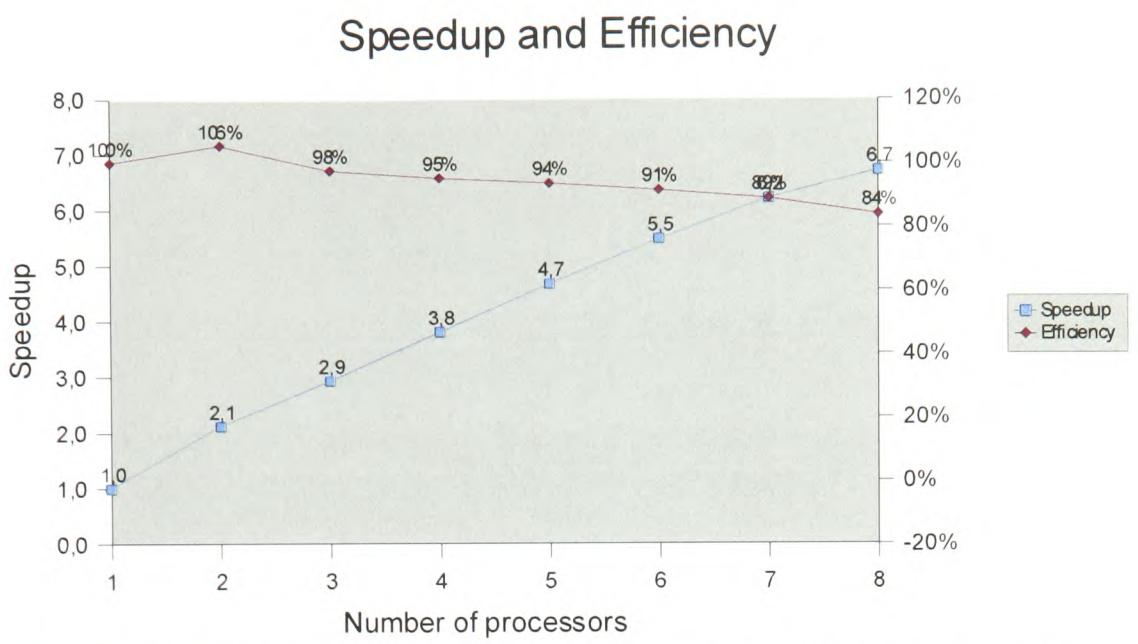


Illustration 6.5.18: Speedup and efficiency results for 1 to 8 processors, running 2000 iterations with JUSTSOLVER Laplace 3D on a 780 blocks EXTV grid.

The super linear speedup achieved from one to two processors is observed in many different Java programs. This reflects the behavior of Java's HotSpot compiler. Using only one processor the profiling task of the HotSpot compiler itself consumes appreciable time to find the program's most time consuming regions (hot spots). ***JUSTSOLVER*** Laplace 3D demonstrates excellent (almost linear) speedup at the hardware configuration utilized.

6 Computational and physics model Validation in JUSTGrid

7 Multipysics Simulation Results with *JUSTGRID*

7.1 JUSTSOLVER Euler3D

To compare the Java based ***JUSTSOLVER*** Euler3D with a flow solver written in 'C' (ParNSS) an EXTV simulation using a free stream value of Mach 8.0 and an angle of attack 0.0 was chosen.

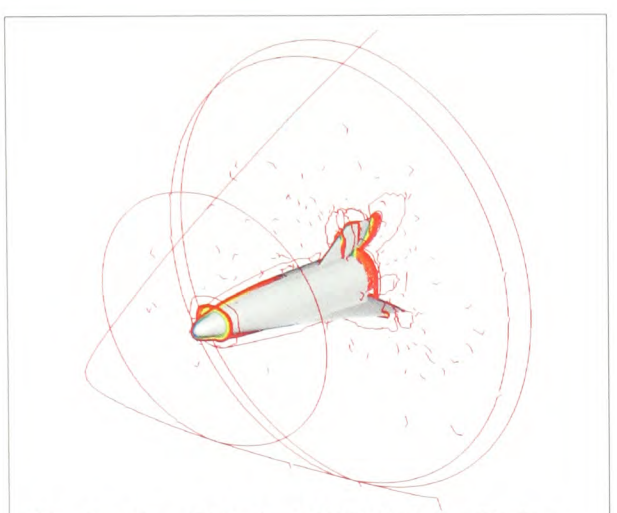


Illustration 7.1.1: JUSTSolver Euler 3D, EXTV, Mach number distribution.

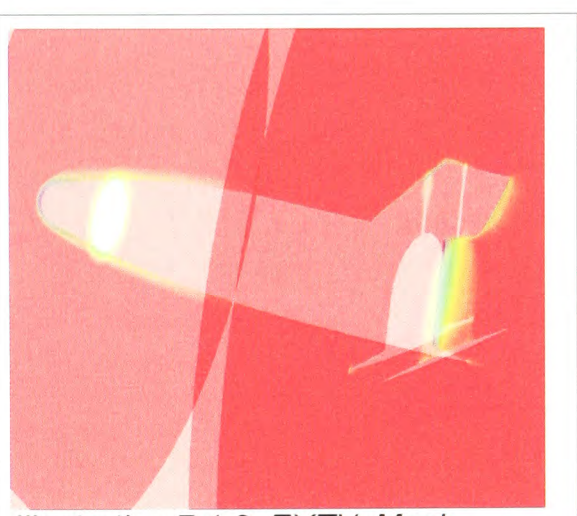


Illustration 7.1.2: EXTV, Mach number distribution on transparent slices.

ParNSS is a legacy 3D structured multiblock code written in 'C'. ParNSS is utilizing the MPI (Message Passing Interface) library to implement the parallelization and communication between neighboring blocks. The implemented numerics for the flux computation (van Leer) for ParNSS and ***JUSTSOLVER*** Euler3D are almost identical (99%) at the source code level. Hence, this provides an excellent opportunity to perform reliable performance comparisons between a 'C' based CFD solver and a Java based flow solver.

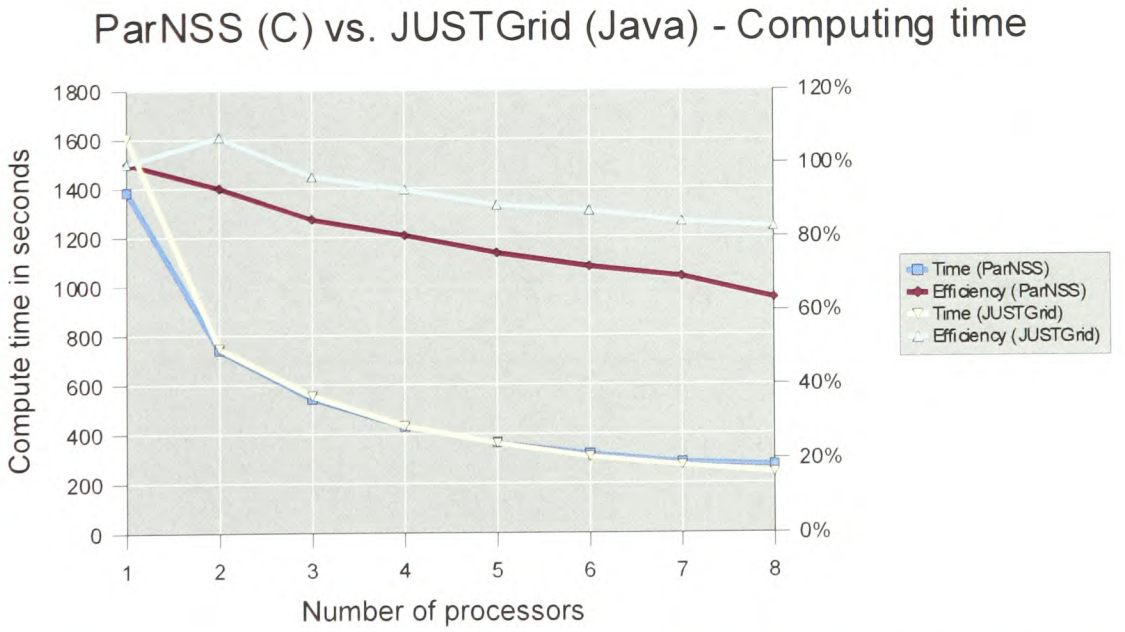


Illustration 7.1.3: Computing time and efficiency results for 1 to 8 processors, running 200 iterations with JUSTSolver Euler 3D and ParNSS on a 780 blocks, 755,300 grid points, 538,752 cells EXTV grid.

7 Multipysics Simulation Results with JUSTGrid

Due to the profiling task of the HotSpot compiler the Java solver is much slower with one processor than the 'C' solver. Employing 5 or more processors ***JUSTSolver*** Euler3D is faster than ParNSS. For 8 processors the time difference is already more than 30 seconds for only 200 iterations.

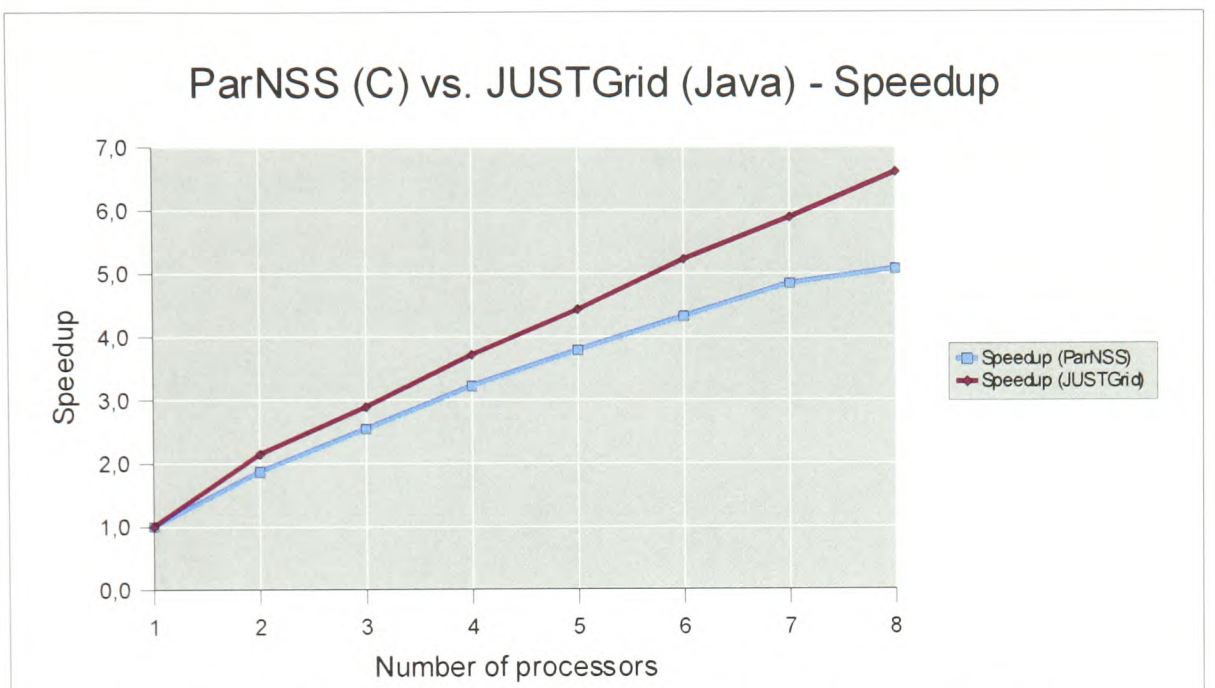


Illustration 7.1.4: Speedup results for 1 to 8 processors, running 200 iterations with JUSTSolver Euler 3D and ParNSS on a 780 blocks, 755,300 grid points, 538,752 cells EXTV grid.

JUSTGrid / JUSTSolver Euler3D achieves better linear speedup than ParNSS. In 8.1.1.3 it is shown that Java programs can achieve linear speedup for numerical applications on large SMP machines with more than 60 processors.

> **Note**: Both computations, Java and C were not able to 100% utilize all 8 processors. This is the reason of the not optimal results for both computations above. In the next chapter the influence of the computational load on the parallel efficiency on current hardware will be shown.

7.2 Magneto Hydro Dynamic (MHD)

7.2.1 Brio-Wu's Shock-Tube

The solution was computed up to time t = 0.25s, because the numerical solution has reached the end of the computational domain. Computational results show excellent agreement with the original results. This shows that the physics and numerics are implemented correctly.

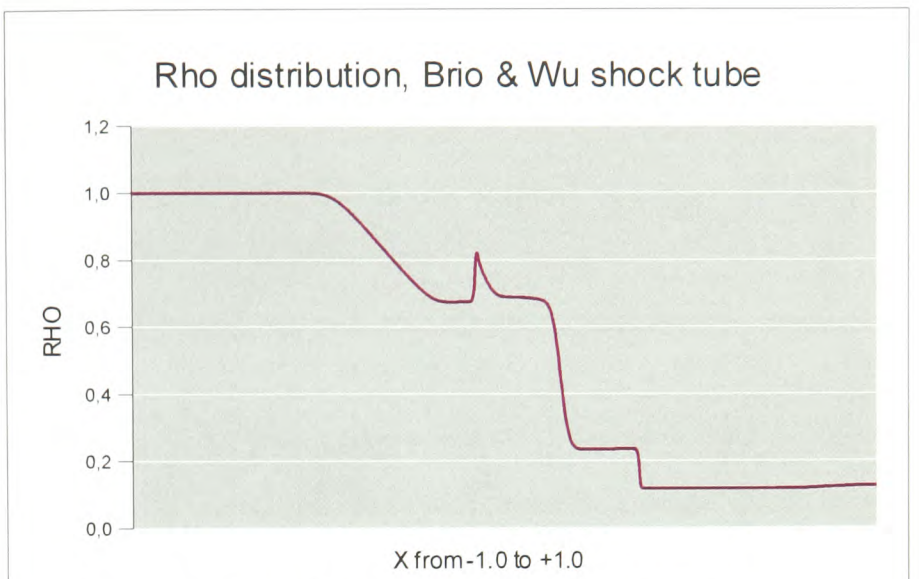


Illustration 7.2.1: 1D MHD solution, rho (density) distribution for well known Brio & Wu shock tube.

7.2.2 MHD 2D test case - Riemann Problem

This 2D Riemann problem was selected from Torrilhon [TRR01].

7.2.2.1 Computational Domain

The computational domain is given by the rectangle [-0.4,0.4;-0.4,0.4].

7.2.2.2 Initial Conditions

initial data ($\boldsymbol{B}_0=\frac{1}{\sqrt{2}}(1,0,0)^T$)				
	$\rho_0(x,y)$	u_x	u_y	$p_0(x,y)$
x<0, y<0	10	0	0	15
otherwise	1	0	0	0,5

7.2.2.3 Boundary Conditions

The boundaries are treated as outflow boundary.

7.2.2.4 Structure of Solution

A solution was computed up time level t = 0.1 s.

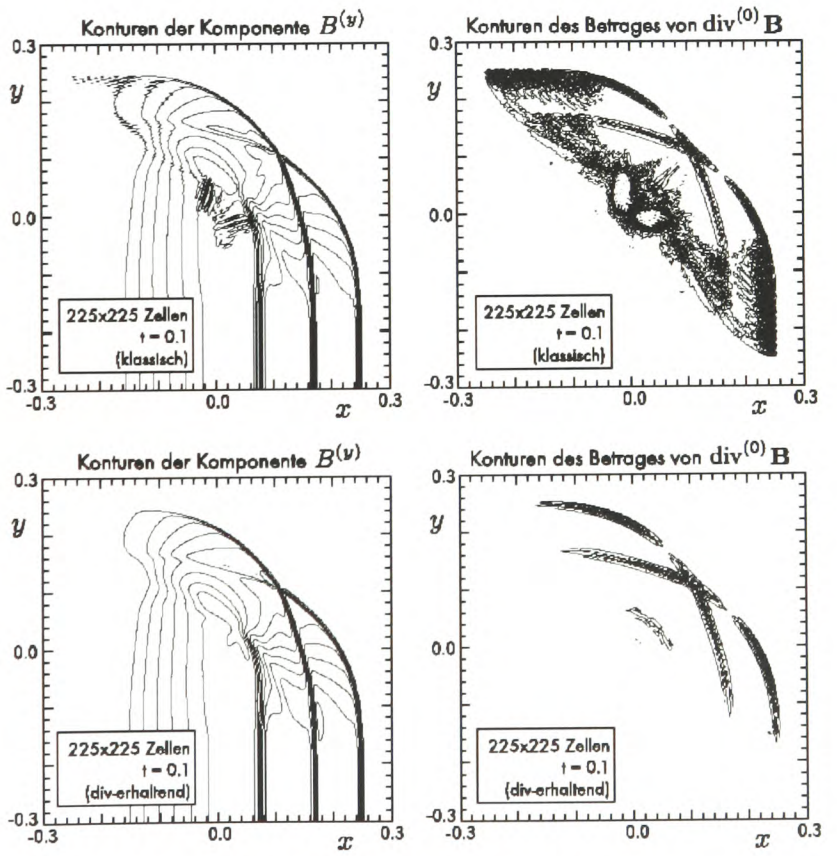


Illustration 7.2.2: The solutions depicted above are a comparison between the classical Finite-Volume (FV) method (first row) and a divergence-conserved FV method (second row). Depicted are the contour of By and the absolute value of the numerical divergence operator for the magnetic induction, div(0).

7 Multipysics Simulation Results with JUSTGrid

The computational grid comprises 300 × 300 cells. In the following computational results from *JUSTSolver MHD Riemann 2D* are presented and compared with the computations of Torillhon.

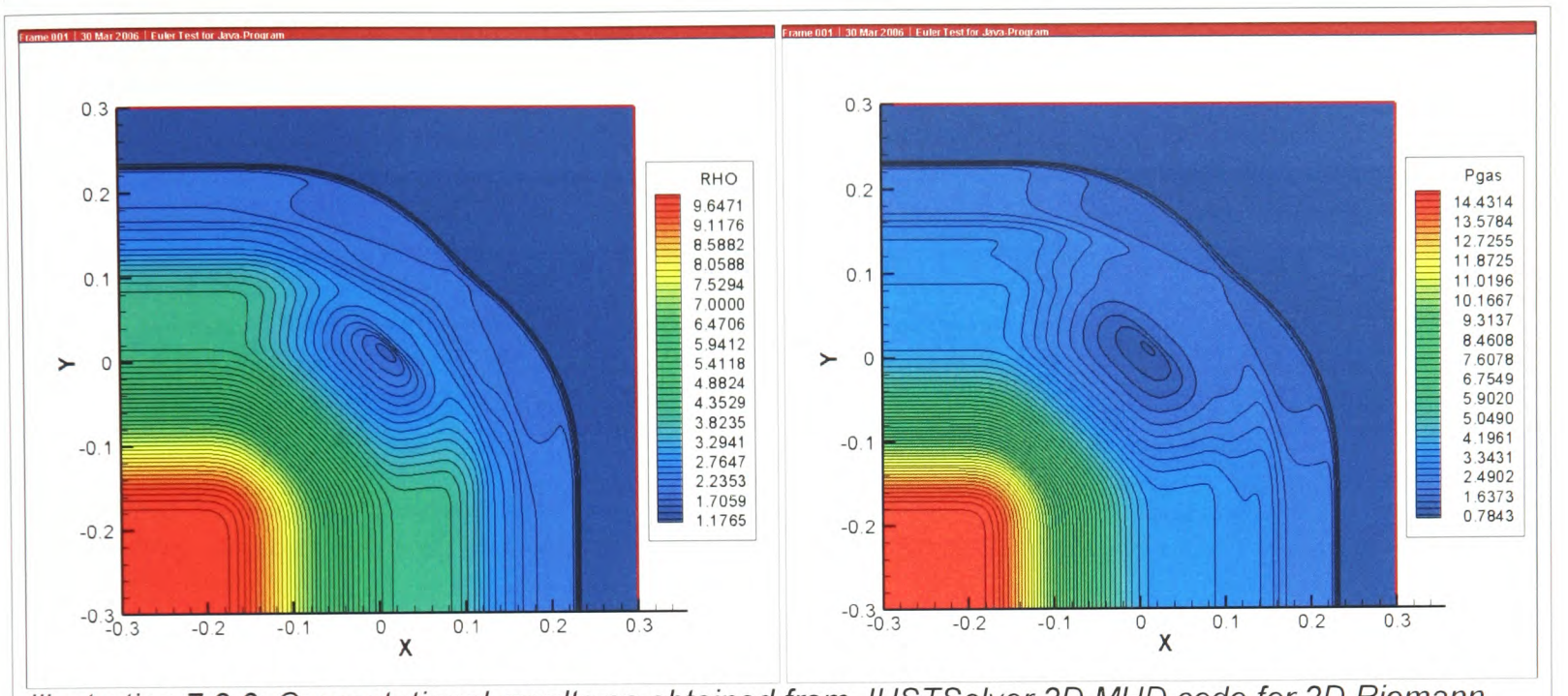


Illustration 7.2.3: Computational results as obtained from JUSTSolver 2D MHD code for 2D Riemann problem: left: density distribution, right: pressure distribution. The results from Torrilhon are shown in Illustration 7.2.3. (grid: 300X300, t=0.1s)

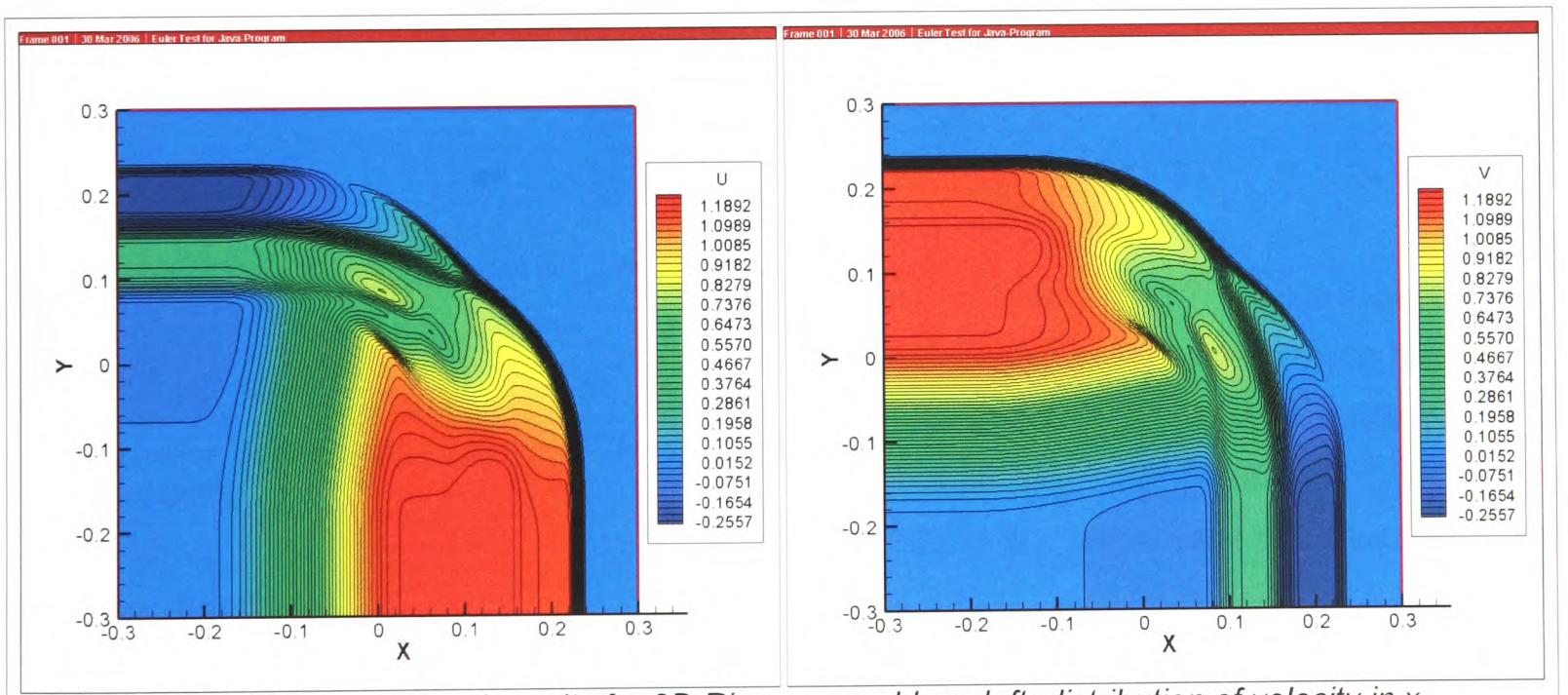


Illustration 7.2.4: Computational results for 2D Riemann problem: left: distribution of velocity in x direction, right: distribution of velocity in y direction. (grid: 300X300, t=0.1s)

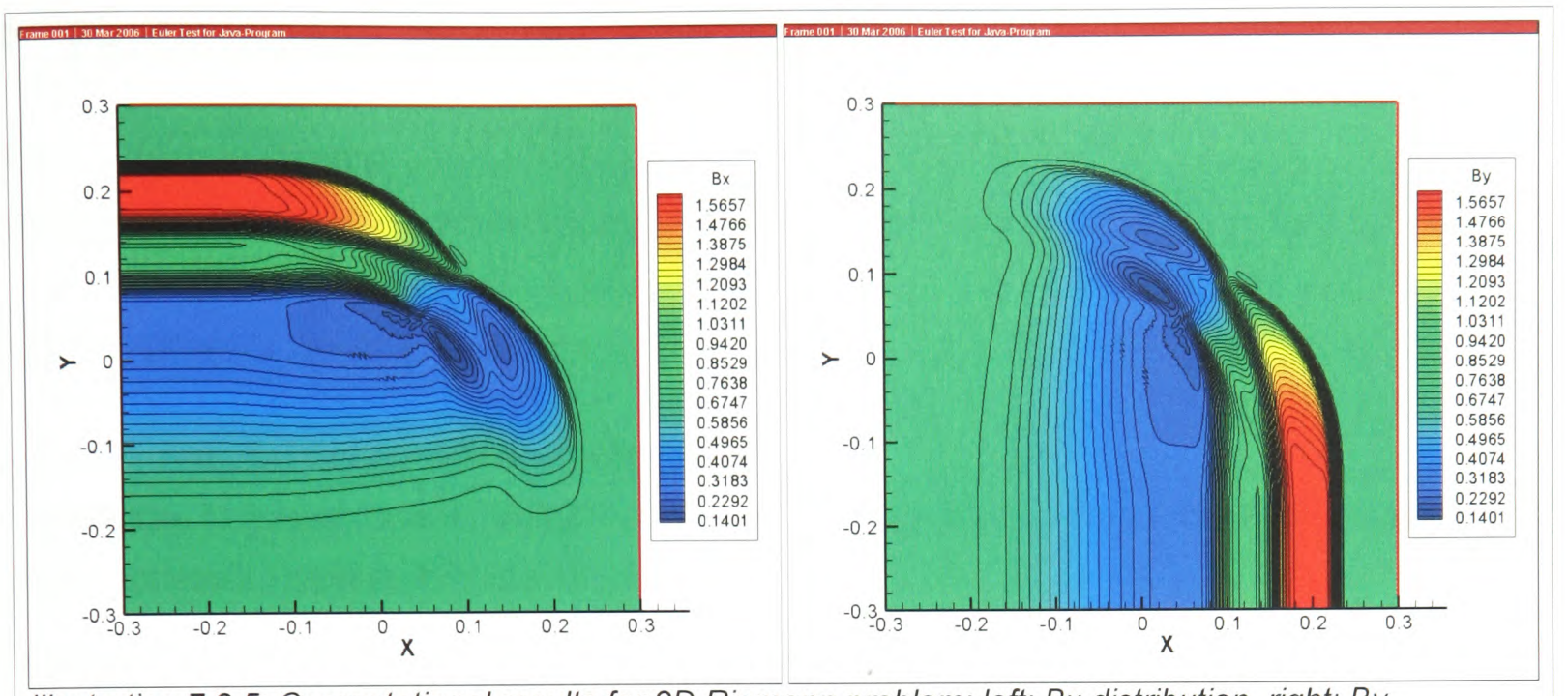


Illustration 7.2.5: Computational results for 2D Riemann problem: left: Bx distribution, right: By distribution. Comparison with Illustration 7.2.2 shows excellent agreement with Torillhon results. (grid: 300 x 300, t=0.1s)

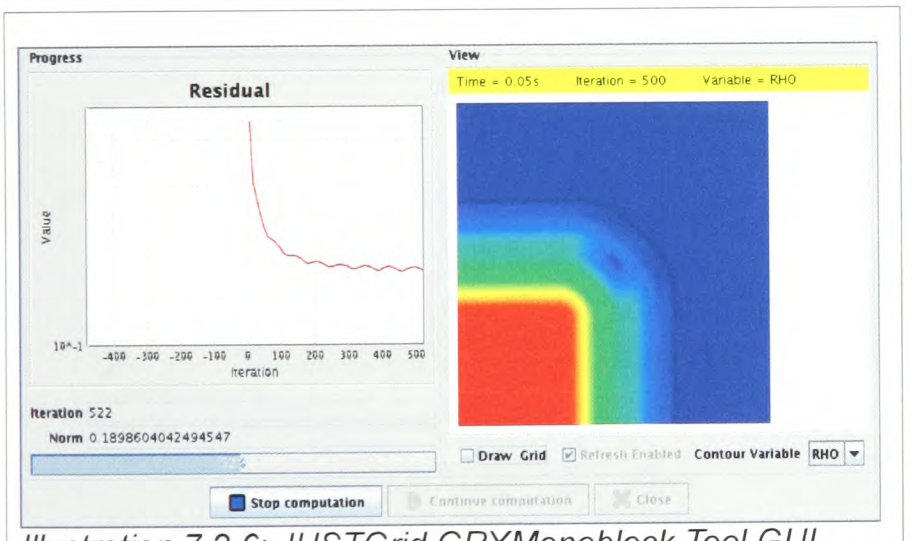


Illustration 7.2.6: JUSTGrid GRXMonoblock Tool GUI, showing online visualization while the 2D MHD Riemann solver is running.

7.2.3 *JUSTGRID*'s GRXMonoblock Tool

The Illustration beside shows the online visualization feature of JUSTGrid's GRXMonoblock Tool. It gives a good impression about the current state of the simulation. Clearly, it is not meant to be a replacement for visualization tools like TecPlot™ or Ensight™. Another useful feature of JUSTGrid's GRXMonoblock Tool is the QuickTime™ movie generation during the simulation.

8 Performance Results with *JUSTGRID*

8.1 Simple Tests

First some simple standard test were done to compare the execution speeds of C++ and Java.

8.1.1 Matrix multiplication

8.1.1.1 Sequential Matrix Multiplication

A sequential (1 thread) matrix multiplication using a 30 times 30 matrix doing 10000 iterations on a single processor Pentium 4 PC running Linux.

Exactly the same source was used for both benchmarks. (C++ and Java)

```
// get start time here
  for( n=0; n<maxIterations; n++)
  {
    for( i=0; i<dim; i++ )
    {
      for( j=0; j<dim; j++ )
      {
        for( k=0; k<dim; k++ )
        {
          c[i][j] += a[i][k]*b[k][j];
        }
      }
    }
  }
// get end time here
```

Runtime (2GHz Pentium 4, 1GB Memory)	1 run	2 run	3 run	4 run	5 run	6 run	7 run	8 run
GNU g++ -O3 -mcpu=pentium4 -march=pentium4 -Wall (Version 3.3.1)	3,15	3,19	3,22	3,16	3,15	3,17	3,16	3,16
Intel icc -O3 -mcpu=pentium4 -march=pentium4 (Version 8.0)	3,23	3,23	3,25	3,23	3,23	3,23	3,23	3,25
Sun Java HotSpot Client VM (Version 1.4.2_02-b03)	3,86	3,88	3,90	3,90	3,90	3,90	3,89	3,90
Sun Java HotSpot Server VM (Version 1.4.2_02-b03)	3,55	3,51	2,12	2,12	2,12	2,12	2,13	2,12

Table 8.1.1: A sequential (1 thread) matrix multiplication using a 30 times 30 matrix doing 10000 iterations on a single processor Pentium 4 PC running Linux.

`a` and `b` are the source and `c` the destination matrix. `dim` and `maxIterations` aren't constant variables so the compilers are not able to do an unroll loop optimization.

The most important result of this benchmark is the enormous speed improvement after the two warmup phases of the Sun Java HotSpot Server VM. This Java runtime version is about 1.5 times faster then the compiled C++ binary.

Due to a Linker error we could not use the `-fast` option with the Intel compiler.

8.1.1.2 Multithreaded Matrix Multiplication

Runtime	time in s
1.1.8_14	516,94
1.2.2_08	38,97
1.3.0_03 Server	37,47
1.3.1_02 Server	21,69
1.4.0_01 Server	19,51
1.4.1_02 Server	17,31
C++ - GCC	26,65

Table 8.1.2: Multithreaded matrix multiplication using a 100 times 100 matrix doing 10000 iterations with 400 threads on a 26 CPU Sun Microsystems Enterprise 6000.

In this configuration the C++ and the Java runtimes are on par.

8.1.1.3 Scaling of a simple numeric benchmark

In this ultra-simple program, many identical threads are used for simple arithmetic computing multiplications and divisions. It is an embarrassingly parallel problem, meaning that the threads do not have to communicate, and thus there is no need for thread synchronization.

The code computes a fixed number of multiplications and divisions and it splits the work among a variable number of threads. These threads then are mapped to the processors by the operating system, relieving the user of the need to employing any kind of message passing library as well as a load balancing algorithm. The code runs on any kind of platform as long as a Java virtual machine is available.

The purpose of this code is to determine whether multi-threading produces a parallel (linear) speedup on the target machine.

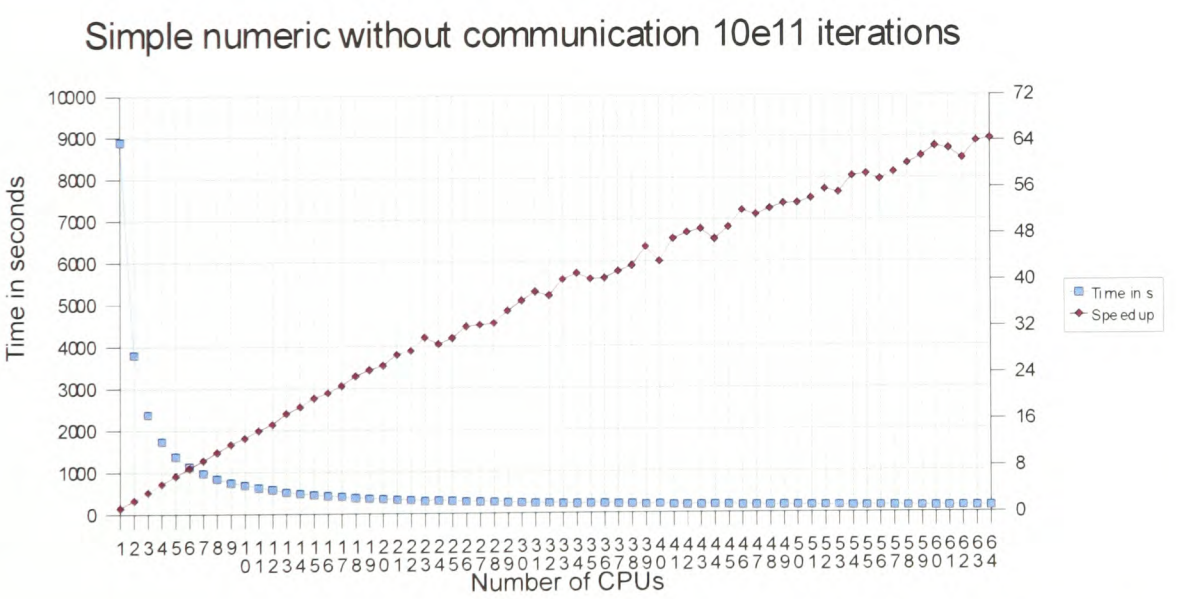


Illustration 8.1.1: Simple numeric benchmark on a Sun Microsystems Enterprise 10000 with 64 UltraSPARC II CPUs and 256GB main memory.

Every benchmark in the single threaded and also in the multi threaded benchmark was done 8 times in the same Java runtime environment.

The performance losses at about every 8 CPUs noticed in Illustration 8.1.1 might be a behavior of the hardware architecture of the Sun Microsystems Enterprise 10000 server.

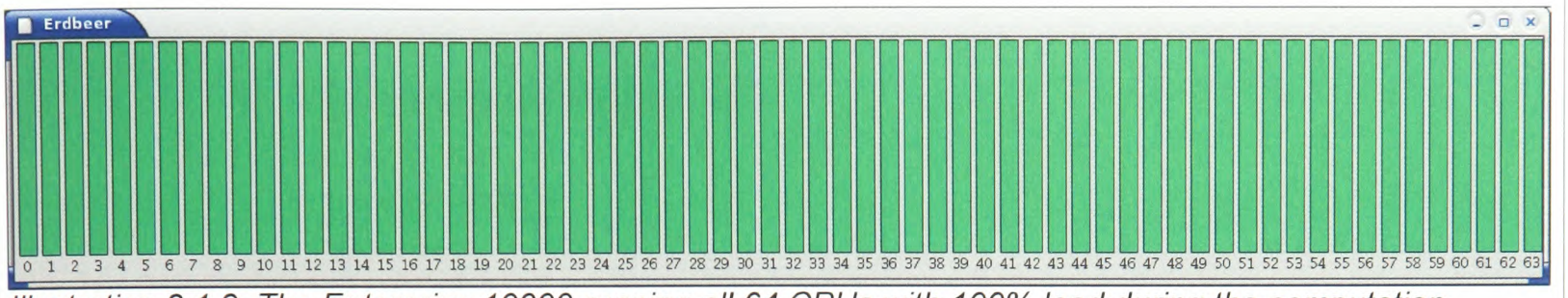


Illustration 8.1.2: The Enterprise 10000 running all 64 CPUs with 100% load during the computation.

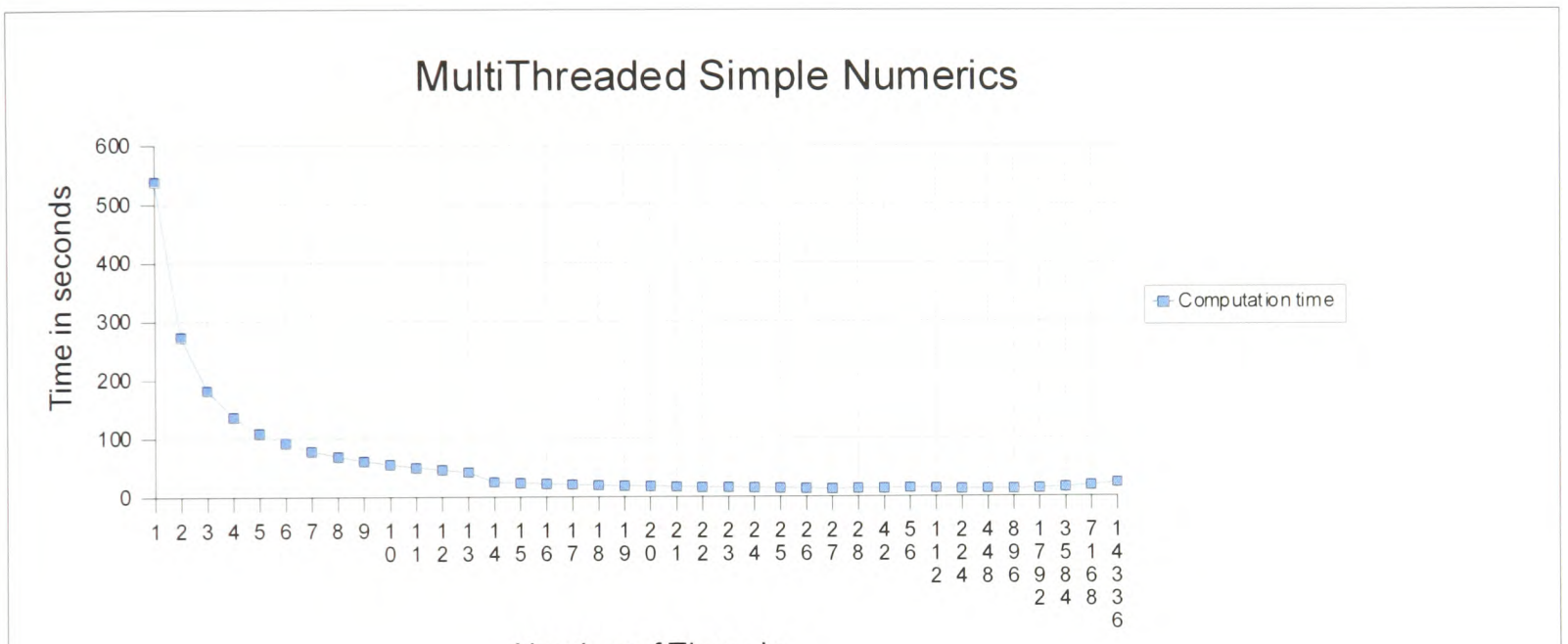


Illustration 8.1.3: This benchmark shows the very small amount of overhead using threads. This benchmark was done on a Sun Microsystems Enterprise 6000 with 28 CPUs.

Illustration 8.1.3 shows that even in case of some 512 threads per CPU the overall computing time rises only slightly.

8.2 Code optimizations and Influence of the computational load on the parallel efficiency

In this chapter we will present performance and efficiency results for **CFD (Computational Fluid Dynamics)** for complex 3D geometries, using the two components of ***JUST***, namely ***JUSTGrid*** and ***JUSTSolver*** on several different multi core computer architectures. The goal is to provide guidelines to achieving best efficiency from modern Java virtual machines (JVM).

All upcoming tests are done on the EXTV grid with 780 blocks, 755,300 grid points containing 538,752 cells without halo cells.

8.2.1 Utilized computer systems

Three different shared memory computer systems where used to run the parallel efficiency tests.

Sun Microsystems - Sun Fire X4440	
Processor type	AMD Opteron 8380, quad core
Processor frequency	2.5GHz
Level 2 cache	4 x 512KB
Level 3 cache	6MB
Total number of processors	4
Total number of cores	16
Main memory	64GB

Sun Microsystems - Sun Fire X4600 m2	
Processor type	AMD Opteron 8384, quad core
Processor frequency	2.7GHz
Level 2 cache	4 x 512KB
Level 3 cache	6MB
Total number of processors	8
Total number of cores	32
Main memory	64GB

Sun Microsystems - Sun Fire T5240	
Processor type	UltraSPARC T2+, 8 cores, 8 hardware threads per core
Processor frequency	1.4GHz
Level 3 cache	4MB
Total number of processors	2
Total number of cores	16
Total number of hardware threads	128
Main memory	64GB

8.2.2 Unoptimized JUSTEuler 3D Code

The JUSTEuler 3D source code was taken from the legacy 'C' based ParNSS code. Over 90% of the Java source code is identical to the 'C' code. Following Kernighans Law “make it right before you make it faster” the code was not optimized for a Java Virtual Machine. With the unoptimized code it was not possible to achieve acceptable results on this modern computer architecture. The code could not fully utilize the available hardware and therefor the parallel efficiency results were not achieved. (See Illustration 8.1.2 on page 129)

8.2.2.1 Benchmark Result - Unoptimized JUSTEuler 3D

OS	Solaris 10 10/08 s10x_u6wos_07b X86
Java Development Kit (JDK)	JDK 1.6.0_13, 32Bit, Server VM, Parallel Garbage Collector
JDK parameter	java -d32 -server -Xcomp -Xnoclassgc -Xms3000m -Xmx3000m -XX:MaxPermSize=512m -XX: +UseFastAccessorMethods -XX:+UseParallelGC eulersolver3d.Main

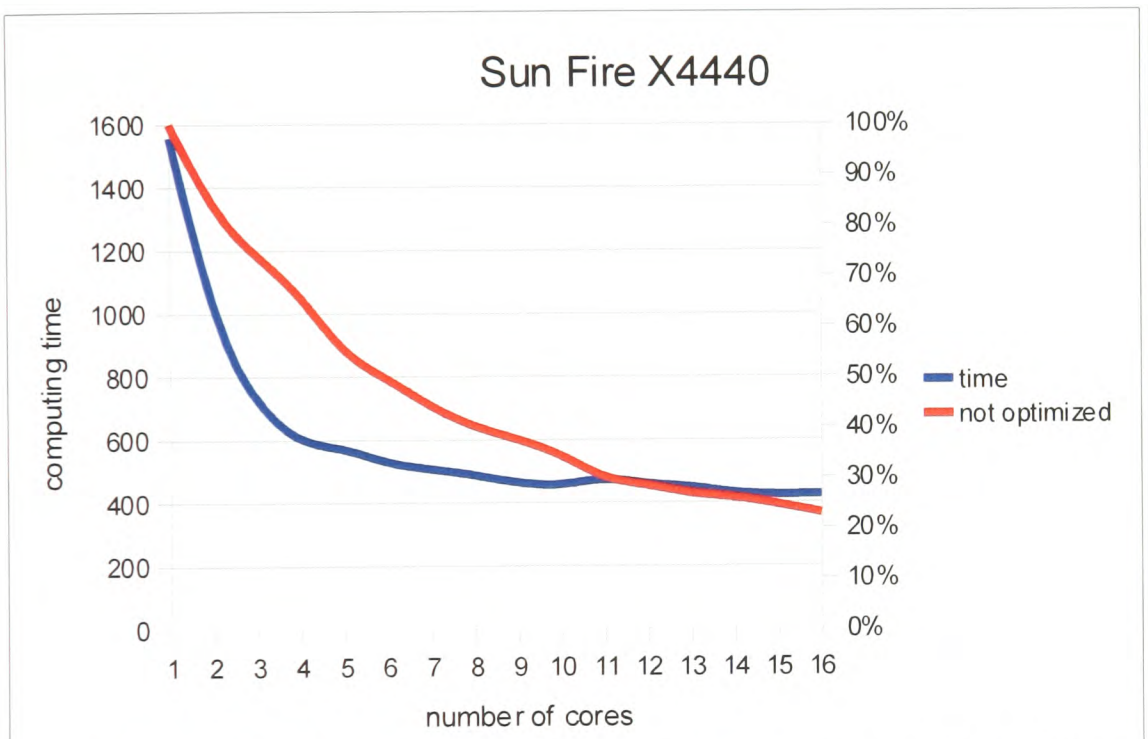


Illustration 8.2.1: Parallel efficiency results for an unoptimized Euler 3D code on a Sun Fire X4440

#cores	time	not optimized
1	1557,879	100,00%
2	950,257	81,97%
3	702,702	73,90%
4	586,772	66,37%
5	575,776	54,11%
6	523,241	49,62%
7	506,846	43,91%
8	488,036	39,90%
9	460,815	37,56%
10	450,652	34,57%
11	482,015	29,38%
12	455,470	28,50%
13	451,240	26,56%
14	426,459	26,09%
15	422,509	24,58%
16	425,275	22,90%

Table 8.2.1: Timing results for an unoptimized Euler 3D code on a Sun Fire X4440

Three years ago, the results where totally different. The overall system performance was much slower and therefore this bad effect was not recognizable.

8.2.3 Optimized JUSTEuler 3D Code

The result of intensive profiling and observing of many environmental metrics of a running Java process (garbage collector, heap memory, eden (?) space, stack, number of running threads...) was that the number of new created objects in the eden space was much too high and the garbage collector created too much load.

With the new Java Visual VM tool, coming with JDK >= Version 1.6.0_07, it was possible to connect to a running Java process and visualize online the important VM metrics.

To minimize the creation of new objects only 8 lines of code where changed in the inner loop of the flux computation and 2 lines in the boundary exchange.

Local variables where changed to instance variables and were locally only set to zero.

Old:

```
method()
{
  Object x = new Object();
  ...
}
```

New:

```
Object x = new Object();

method()
{
  x.setZero();
  ...
}
```

The reusing, instead of new creation of objects has a dramatic impact on the numerical load and the garbage collector, because the amount of dynamic heap access was nearly eliminated. For instance, on the Apple Mac Book Pro the optimized code is about 10 times faster than the unoptimized code. On the Sun Fire X4440 the code is about 4 times faster.

8.2.3.1 Benchmark Results - Optimized JUSTEuler 3D

OS	Solaris 10 10/08 s10x_u6wos_07b X86
Java Development Kit (JDK)	JDK 1.6.0_13, 32Bit, Server VM, Parallel Garbage Collector
JDK parameter	java -d32 -server -Xcomp -Xnoclassgc -Xms3000m -Xmx3000m -XX:MaxPermSize=512m -XX:+UseFastAccessorMethods -XX:+UseParallelGC eulersolver3d.Main

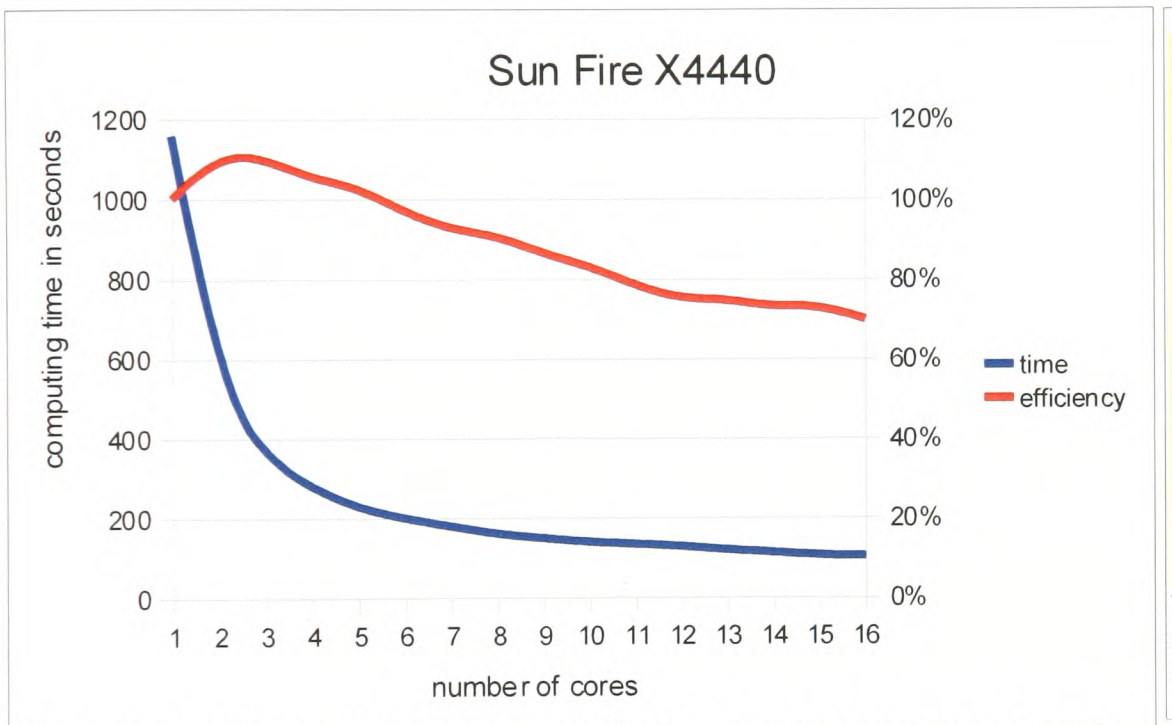


Illustration 8.2.2: Parallel efficiency results for an optimized Euler 3D code on a Sun Fire X4440

#cores	time	efficiency
1	1158,323	100,00%
2	517,878	111,83%
3	349,898	110,35%
4	275,349	105,17%
5	224,799	103,05%
6	199,455	96,79%
7	179,093	92,40%
8	159,229	90,93%
9	149,135	86,30%
10	139,114	83,26%
11	134,425	78,34%
12	129,033	74,81%
13	118,692	75,07%
14	114,032	72,56%
15	104,754	73,72%
16	103,737	69,79%

Table 8.2.2: Timing results for an optimized Euler 3D code on a Sun Fire X4440

With the simply optimized Euler code the computation was 4 times faster on all 16 cores compared to the unoptimized code. Cpu utilization was only 70%-80% per core, therefore overall parallel efficiency was about 70% only.

8.3 *Additional Computational Load*

The computational load of the selected Euler 3D problem was too small to fully utilize the available computing power. For benchmarking purposes, fluxes across faces were computed multiple times. The variable „load" gives the number of sub loops during one iteration.

8.3.1 Benchmark Results - Load efficiency on Sun T5240

The Sun Microsystems, Sun T5240 was not designed for High Performance Computing, but it can be an acceptable system for problems where the number of processors is more important than the single thread performance. For 2010, the Sun Niagara 3 chip is expected with 16 cores and 16 hardware threads on each core. With 256 hardware threads on each processor the 8 processor Niagara 3 system, also expected for 2010, will have 2048 hardware threads and each of theses hardware threads will act like a general purpose processor.

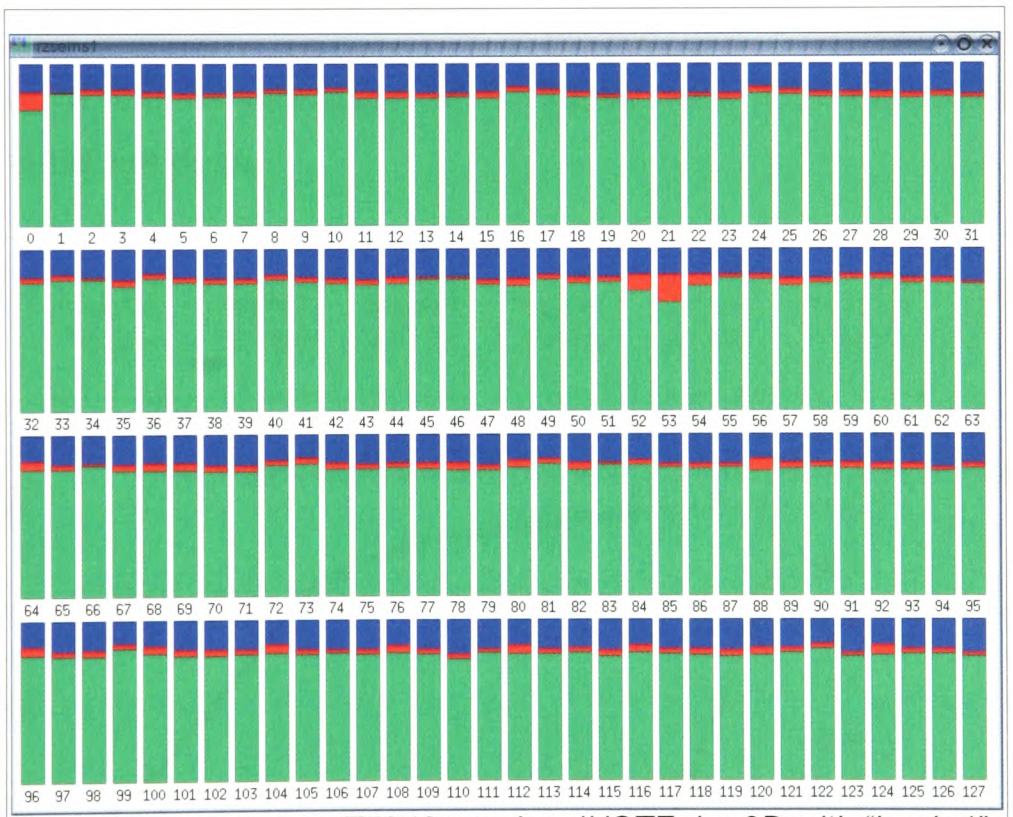


Illustration 8.3.1: Sun T5240 running JUSTEuler 3D with "load=1"

perfbar cpu monitor: green=user usage, red=kernel usage, blue=idle.

Even on the T5240 with a modestsingle thread hardware performance the simulation was not big enough to fully utilize the available cpu resources.

8 Performance Results with JUSTGrid

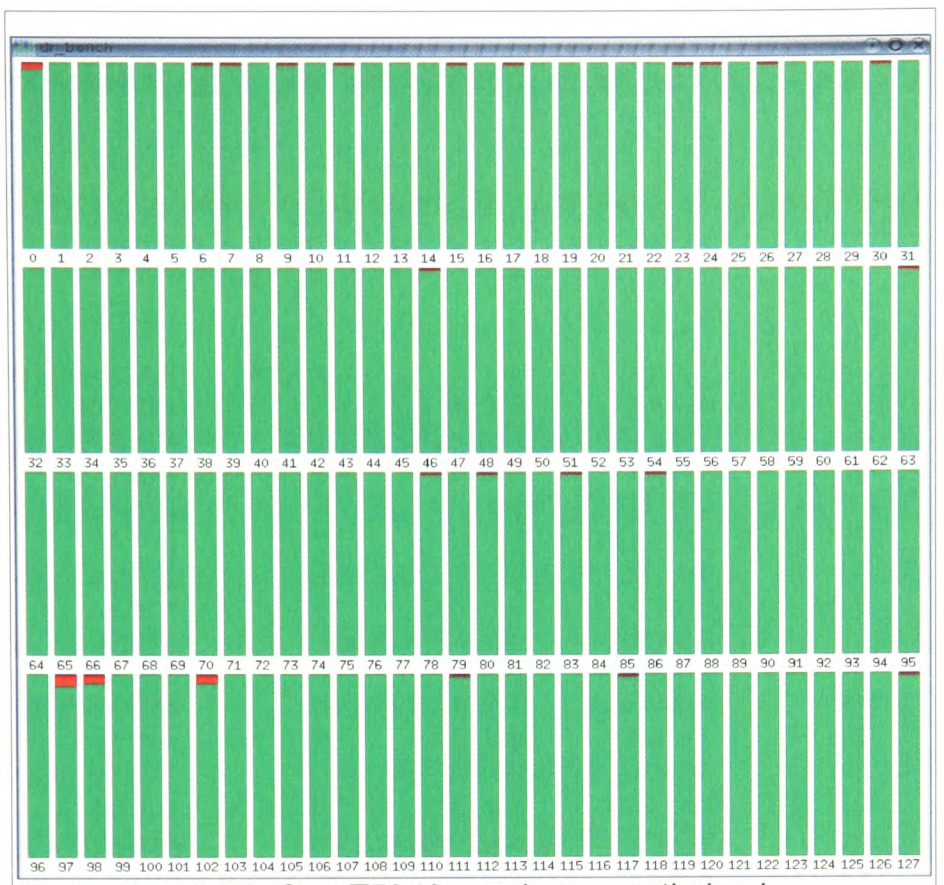


Illustration 8.3.2: Sun T5240 running an optimized JUSTEuler 3D with "load=100"

Assuming with a load of 100 the processors of the T5240 were 100% utilized the computations for the following load values are done.

load	1	2	5	10	20	50	100	
computing time	144,243	200,372	331,856	612,516	1187,706	2894,877	5785,568	seconds
load efficiency	40,11%	57,75%	87,17%	94,46%	97,42%	99,93%	100,00%	

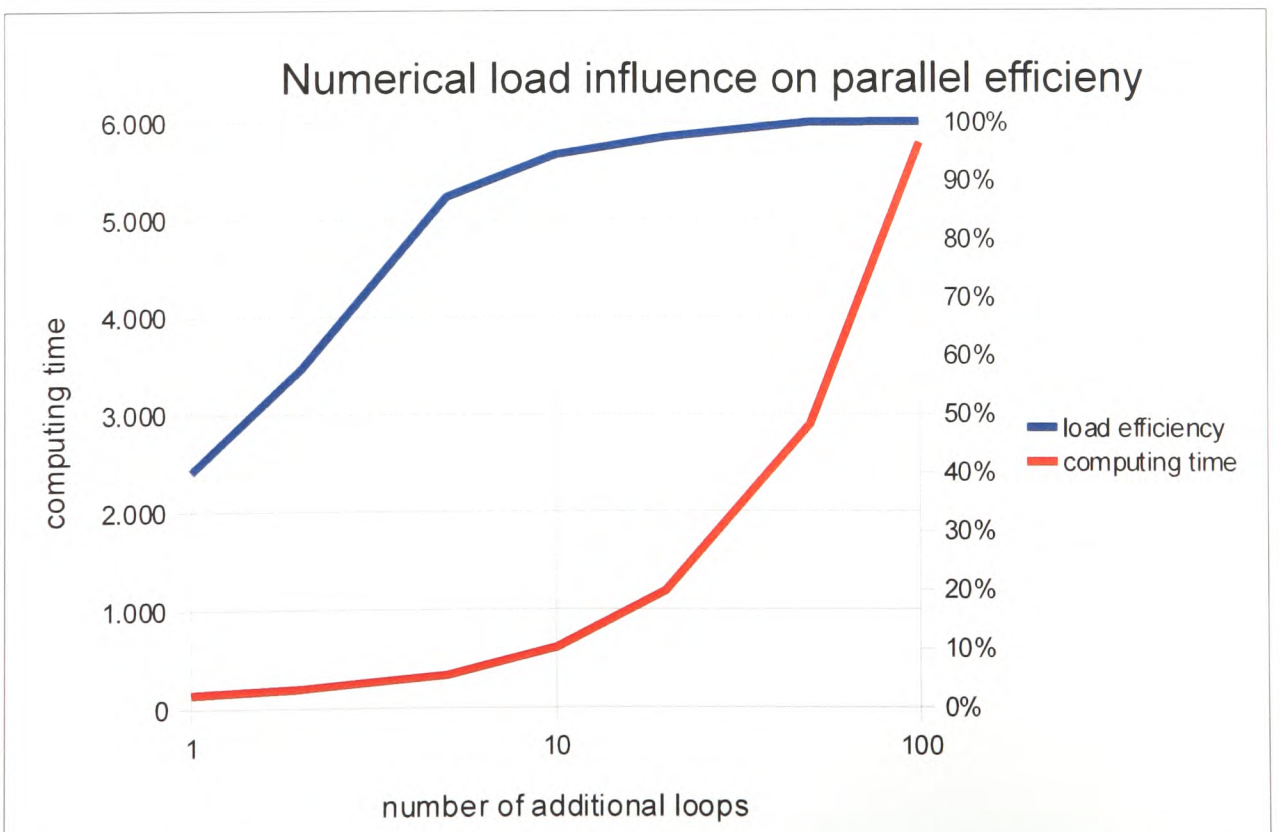


Illustration 8.3.3: The illustration shows, increasing computational load by a factor of 20, that utilization level was already at more than 90%.

8.3.2 Benchmark Results for different numerical load on a Sun Fire X4440

Timing and parallel efficiency results for different load values on an optimized Euler code inclusive one benchmark on an unoptimized Euler code.

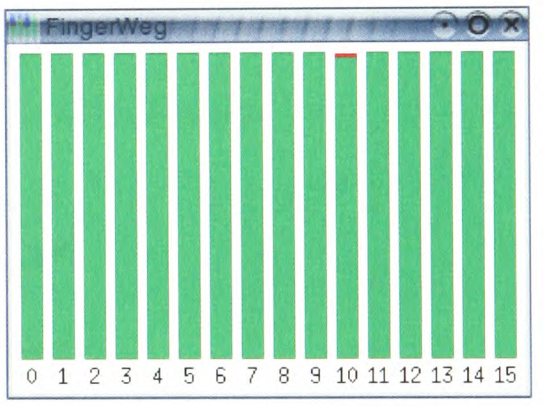
#cores	time	not optimized
1	1557,879	100,00%
2	950,257	81,97%
3	702,702	73,90%
4	586,772	66,37%
5	575,776	54,11%
6	523,241	49,62%
7	506,846	43,91%
8	488,036	39,90%
9	460,815	37,56%
10	450,652	34,57%
11	482,015	29,38%
12	455,470	28,50%
13	451,240	26,56%
14	426,459	26,09%
15	422,509	24,58%
16	425,275	22,90%

#cores	time	load 1
1	1158,323	100,00%
2	517,878	111,83%
3	349,898	110,35%
4	275,349	105,17%
5	224,799	103,05%
6	199,455	96,79%
7	179,093	92,40%
8	159,229	90,93%
9	149,135	86,30%
10	139,114	83,26%
11	134,425	78,34%
12	129,033	74,81%
13	118,692	75,07%
14	114,032	72,56%
15	104,754	73,72%
16	103,737	69,79%

#cores	time	load 5
1	3342,463	100,00%
2	1429,317	116,93%
3	932,371	119,50%
4	697,295	119,84%
5	580,354	115,19%
6	490,003	113,69%
7	425,291	112,27%
8	374,745	111,49%
9	339,593	109,36%
10	314,488	106,28%
11	289,398	105,00%
12	273,788	101,74%
13	244,984	104,95%
14	234,387	101,86%
15	222,700	100,06%
16	204,501	102,15%

#cores	time	load 20
1	10051,070	100,00%
2	5787,959	86,83%
3	3345,290	100,15%
4	2411,630	104,19%
5	1936,689	103,80%
6	1582,043	105,89%
7	1354,922	105,97%
8	1197,699	104,90%
9	1078,350	103,56%
10	976,070	102,97%
11	880,764	103,74%
12	825,282	101,49%
13	745,982	103,64%
14	705,805	101,72%
15	635,915	105,37%
16	618,387	101,59%

#cores	time	load 100
1	55317,510	100,00%
2	27708,375	99,82%
3	14934,260	123,47%
4	11334,756	122,01%
5	8934,050	123,84%
6	7419,530	124,26%
7	6497,718	121,62%
8	5711,156	121,07%
9	5039,137	121,97%
10	4524,777	122,25%
11	4114,497	122,22%
12	3780,712	121,93%
13	3481,365	122,23%
14	3109,710	127,06%
15	3000,380	122,91%
16	2839,896	121,74%



The last illustration shows the Solaris perfbar tool on the fully utilized system.

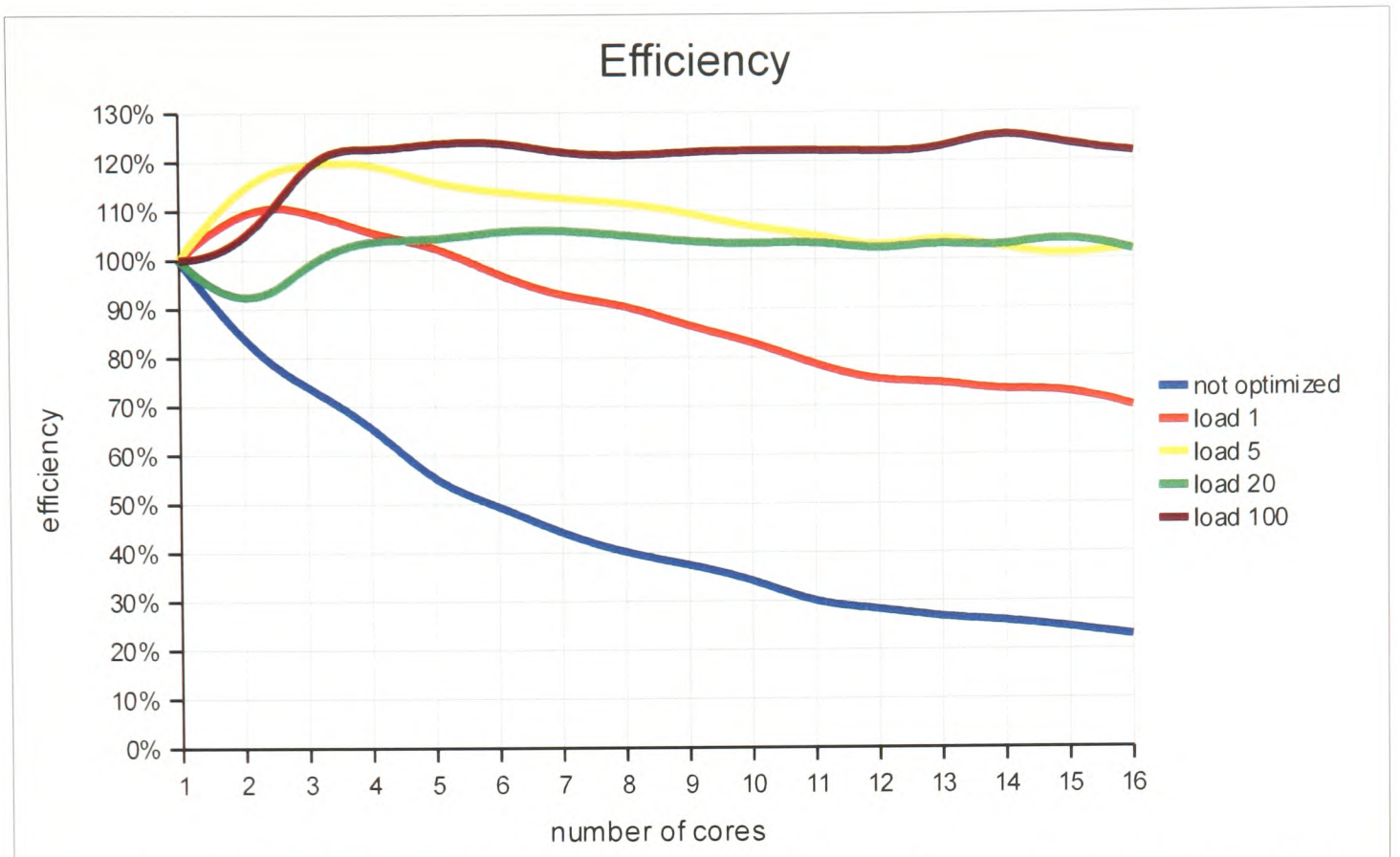


Illustration 8.3.4: Efficiency gains from solver optimization and increased computational load

As we can see in the combined efficiency diagram there is a lot of influence on the computation coming from the OS memory management and the Java HotSpot compiler. This is the reasons why the computation with 1 to 2 cores differs substantially from the rest.

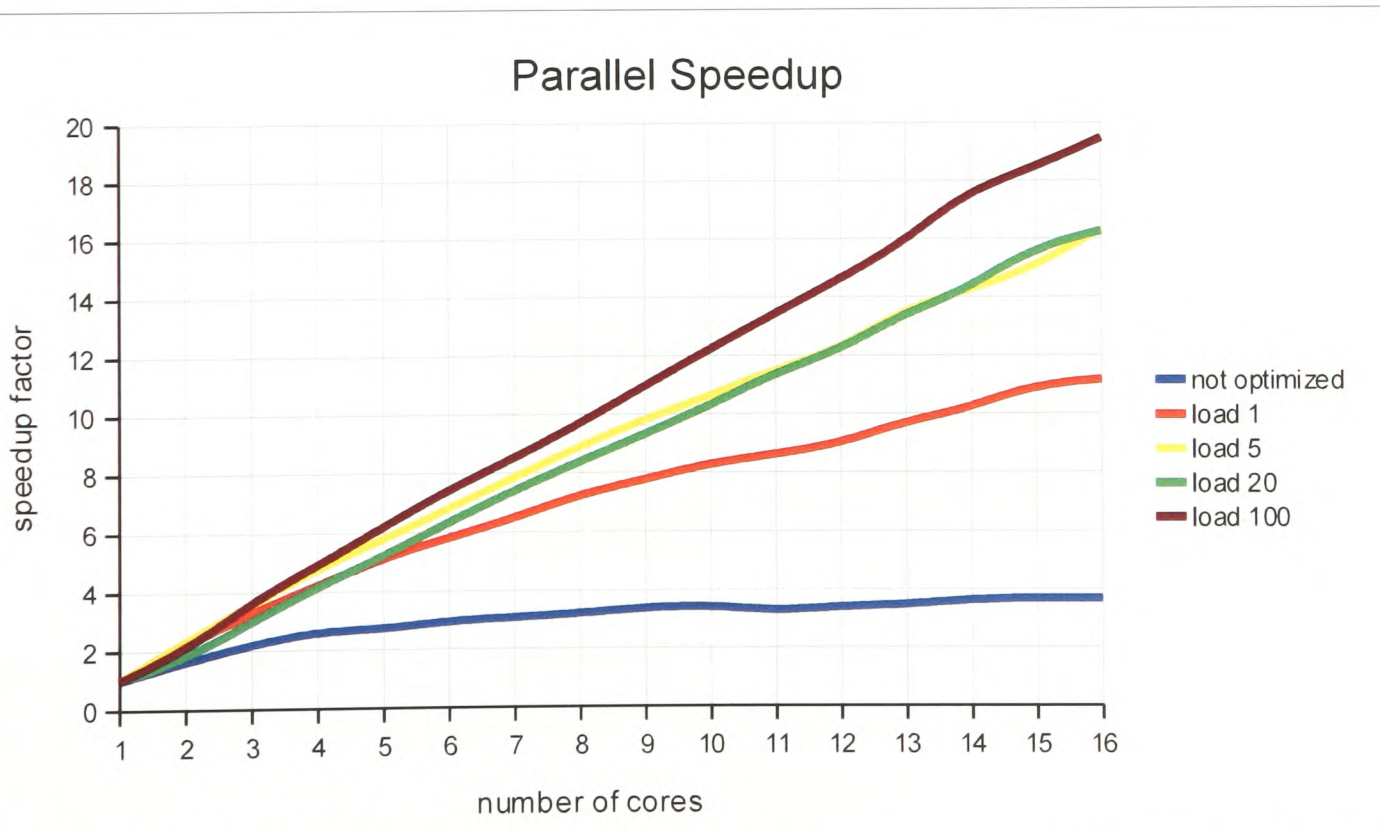


Illustration 8.3.5: Parallel speedup gains from solver optimization and increased computational load

Because of the improper starting point with one core, due to the behavior of the HotSpot compiler, a maximum super linear speedup of about 20 was achieved with 16 cores only.

8 Performance Results with JUSTGrid

To rener the speedup diagram more meaningful, I chose only the values for 4,8,12 and 16 cores. In this case, the maximum linear speedup is 16 and this maximum value was actually achieved.

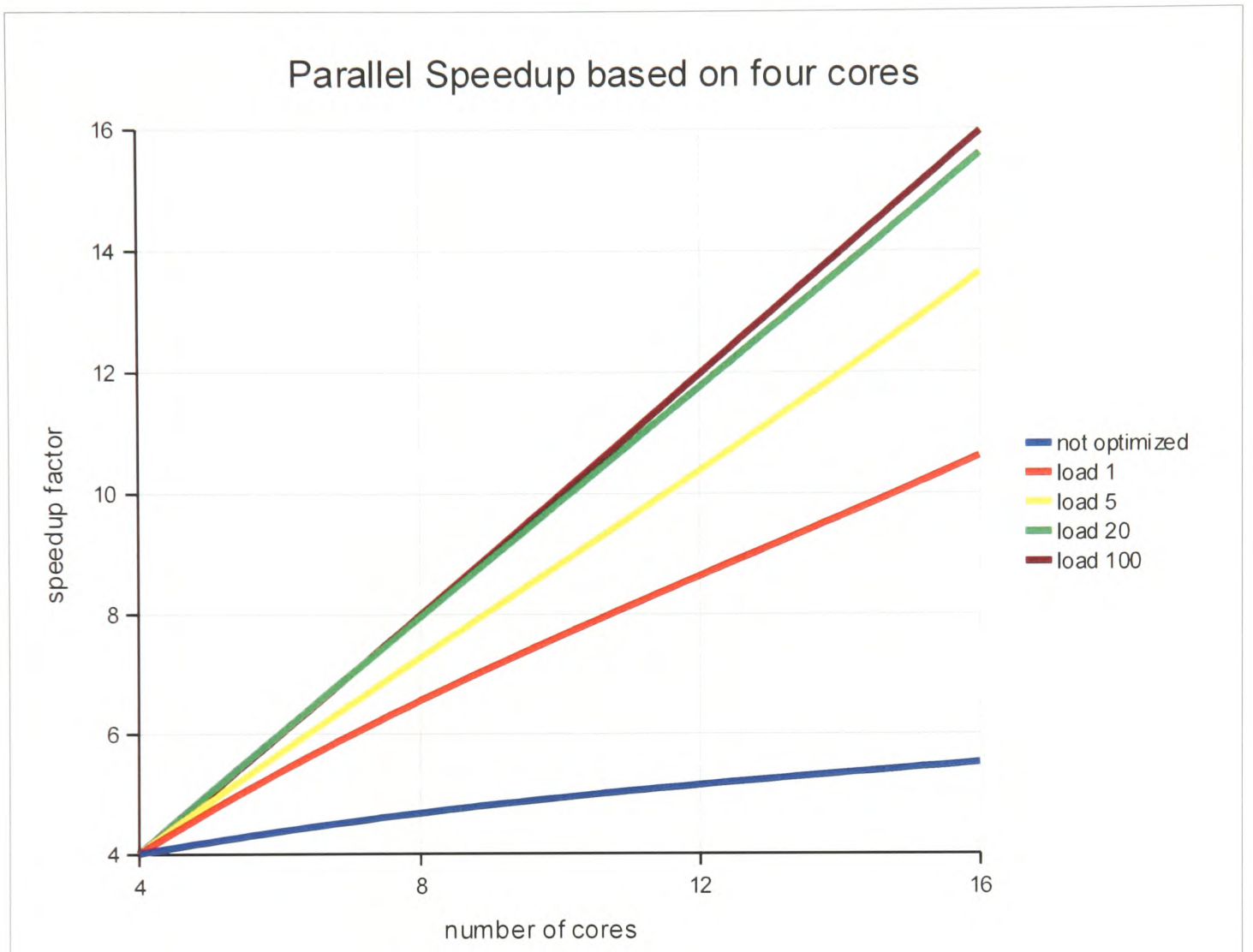


Illustration 8.3.6: Parallel speedup gains from solver optimization and increased computational load based on four cores.

#cores	4	8	12	16
not optimized	4,00	4,81	5,15	5,52
load 1	4,00	6,92	8,54	10,62
load 5	4,00	7,44	10,19	13,64
load 20	4,00	8,05	11,69	15,60
load 100	4,00	7,94	11,99	15,97

Without this measure, when using less than four cores, realistic linear speedups were achieved and the influence of the additional load was perfectly demonstrated.

8.3.3 Benchmark Results for different numerical load on a Sun Fire X4600 m2

After the X4440 benchmarks a SUN X4600 with 32 cores was available. In addition, a new Solaris version was available. A load value of 200 was needed to fully utilize this system. This configuration twice as fast as the older X4440.

OS	Solaris 10 05/09 s10x_u7 X86
Java Development Kit (JDK)	JDK 1.6.0_13, 32Bit, Server VM, Parallel Garbage Collector
JDK parameter	java -d32 -server -Xcomp -Xnoclassgc -Xms3000m -Xmx3000m -XX:MaxPermSize=512m -XX: +UseFastAccessorMethods -XX:+UseParallelGC eulersolver3d.Main

#cores	time	efficiency	speedup
1	78528,693	100,00%	1,00
2	43730,783	89,79%	1,80
3	27539,026	95,05%	2,85
4	21604,411	90,87%	3,63
5	16367,433	95,96%	4,80
6	13666,323	95,77%	5,75
7	11580,088	96,88%	6,78
8	10523,193	93,28%	7,46
9	9026,869	96,66%	8,70
10	8235,249	95,36%	9,54
11	7448,740	95,84%	10,54
12	6742,094	97,06%	11,65
13	6234,053	96,90%	12,60
14	5854,603	95,81%	13,41
15	5489,885	95,36%	14,30
16	5125,938	95,75%	15,32
17	4763,481	96,97%	16,49
18	4593,263	94,98%	17,10
19	4323,053	95,61%	18,17
20	4108,359	95,57%	19,11
21	3870,309	96,62%	20,29
22	3749,976	95,19%	20,94
23	3624,702	94,20%	21,66
24	3390,473	96,51%	23,16
25	3229,594	97,26%	24,32
26	3186,089	94,80%	24,65
27	3009,084	96,66%	26,10
28	2919,952	96,05%	26,89
29	2834,210	95,54%	27,71
30	2805,211	93,31%	27,99
31	2693,924	94,03%	29,15
32	2568,488	95,54%	30,57

Table 8.3.1: Fully utilized Sun Fire X4600 system with load=200

As was already visible on the 16 cores Sun Fire X4440, the results with 1 and 2 cores differs substantially from the subsequent results because of the Java HotSpot VM engine. Therefore only the results based on 4 cores should be used for comparison.

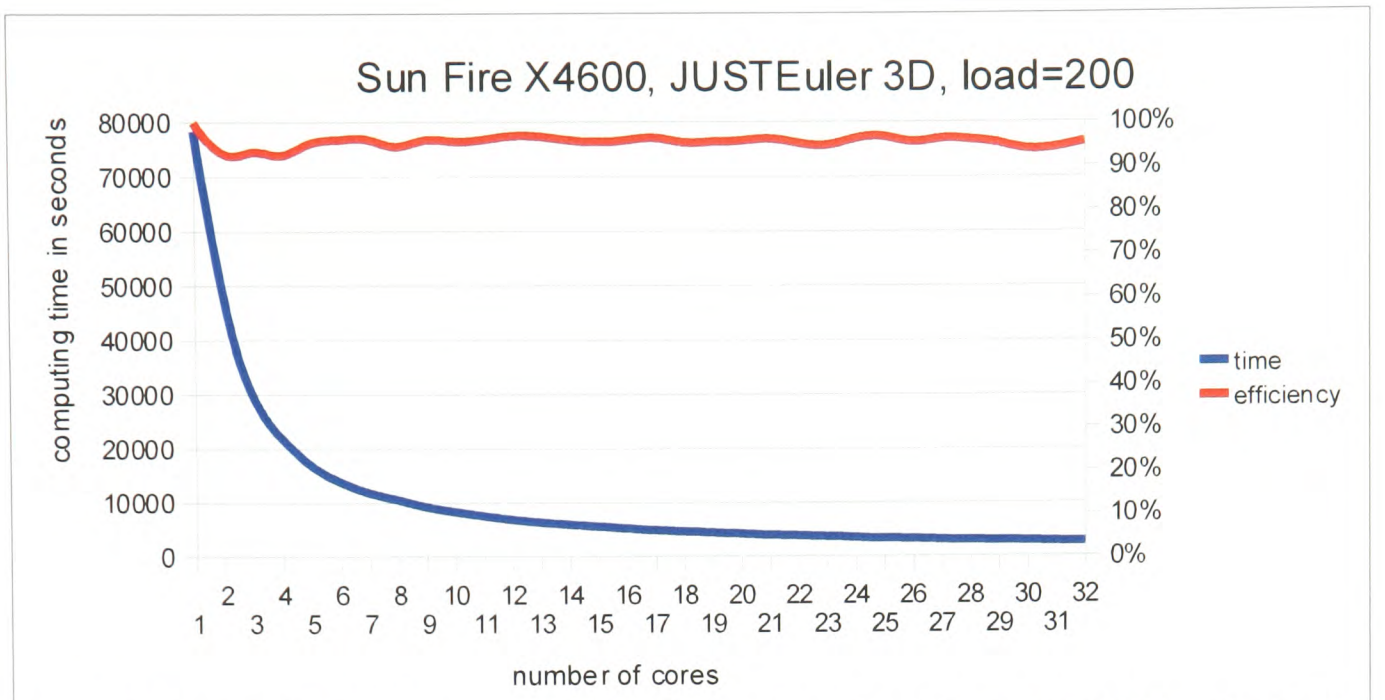


Illustration 8.3.7: Timings and parallel efficiency results for a Sun Fire X4600, optimized Euler solver, load=200.

8.3.3.1 Efficiency gains from increased computational load based on 4 cores

load 1

#cores	time	efficiency	speedup
4	287,543	100,00%	4,00
8	172,602	83,30%	6,66
12	134,482	71,27%	8,55
16	114,842	62,60%	10,02
20	110,059	52,25%	10,45
24	102,916	46,57%	11,18
28	97,945	41,94%	11,74
32	98,849	36,36%	11,64

load 5

#cores	time	efficiency	speedup
4	684,280	100,00%	4,00
8	370,866	92,25%	7,38
12	265,197	86,01%	10,32
16	214,172	79,88%	12,78
20	184,546	74,16%	14,83
24	153,222	74,43%	17,86
28	140,169	69,74%	19,53
32	138,947	61,56%	19,70

load 20

#cores	time	efficiency	speedup
4	2311,097	100,00%	4,00
8	1110,819	104,03%	8,32
12	761,313	101,19%	12,14
16	586,031	98,59%	15,77
20	484,966	95,31%	19,06
24	418,592	92,02%	22,08
28	367,783	89,77%	25,14
32	319,170	90,51%	28,96

load 100

#cores	time	efficiency	speedup
4	10458,978	100,00%	4,00
8	5120,783	102,12%	8,17
12	3517,552	99,11%	11,89
16	2645,301	98,84%	15,82
20	2103,597	99,44%	19,89
24	1736,162	100,40%	24,10
28	1502,180	99,46%	27,85
32	1325,809	98,61%	31,56

load 200

#cores	time	efficiency	speedup
4	21604,411	100,00%	4,00
8	10523,193	102,65%	8,21
12	6742,094	106,81%	12,82
16	5125,938	105,37%	16,86
20	4108,359	105,17%	21,03
24	3390,473	106,20%	25,49
28	2919,952	105,70%	29,60
32	2568,488	105,14%	33,65

8 Performance Results with JUSTGrid

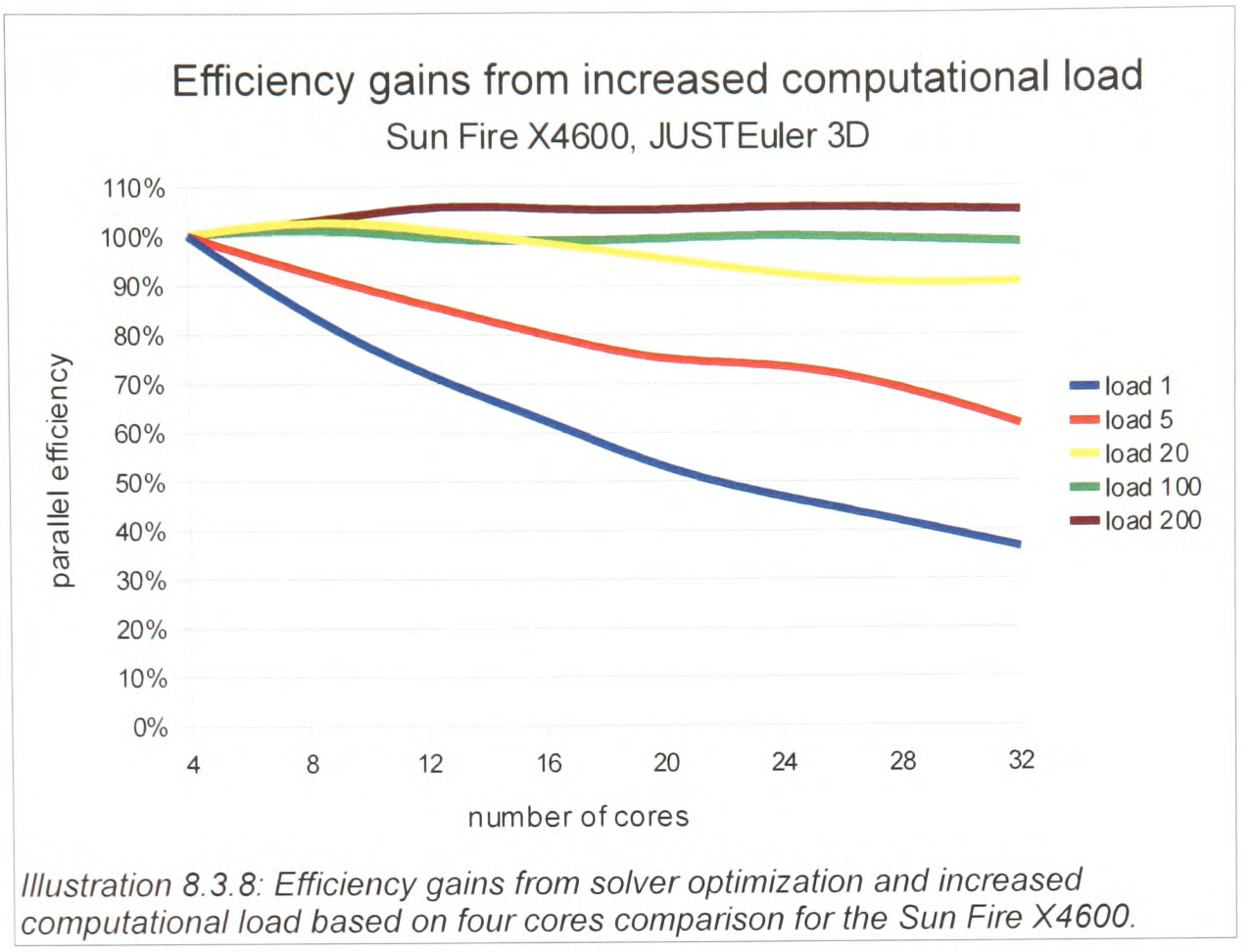


Illustration 8.3.8: Efficiency gains from solver optimization and increased computational load based on four cores comparison for the Sun Fire X4600.

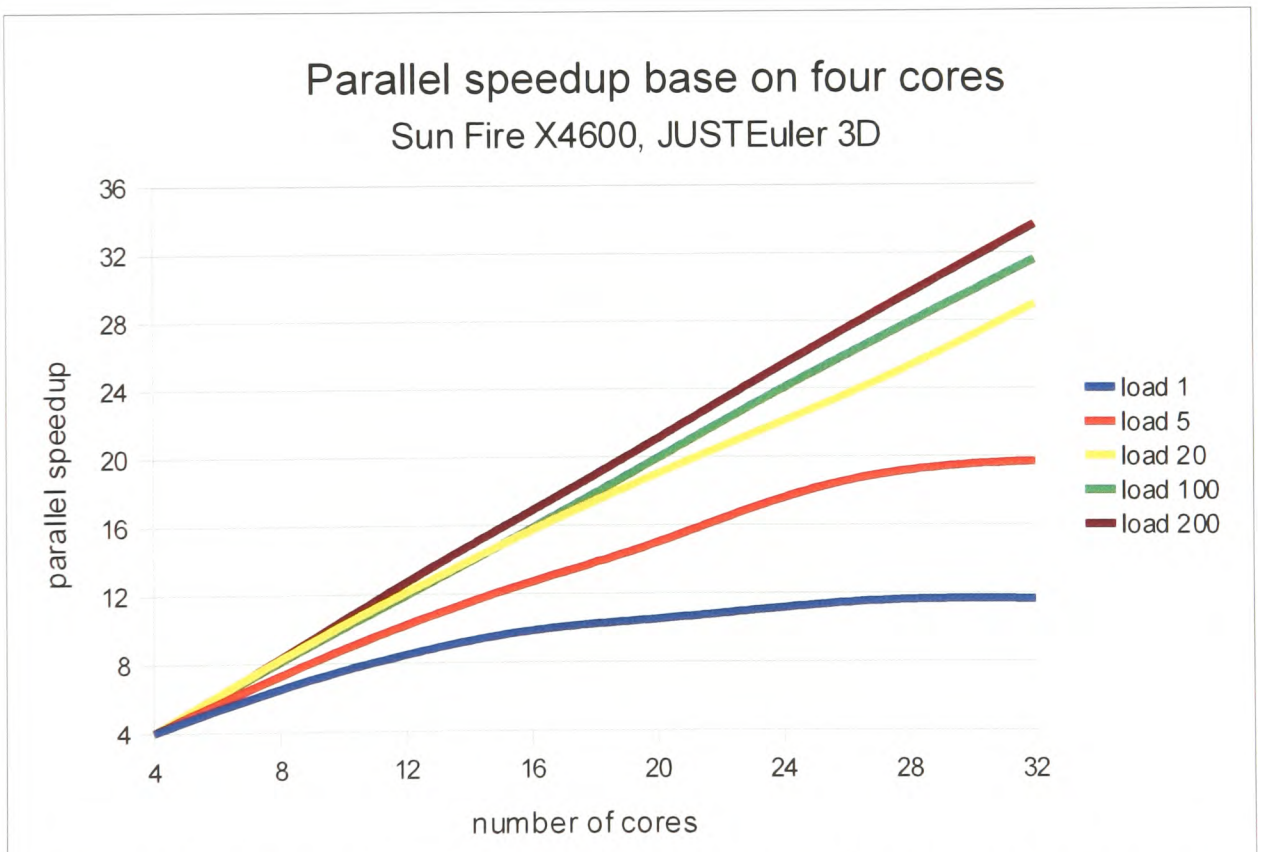


Illustration 8.3.9: Parallel speedup gains from solver optimization and increased computational load based on four cores for the Sun Fire X4600.

These results shows that ***JUSTGrid*** is able to achieve perfect linear speedup on modern multi-core systems under the right conditions. As it was shown, the simple Euler Solver produces not enough computational load to fully utilize such type of systems. But additional numerical load can be generated by more complex geometries or more costly numerical problems, e.g. Navier-Stokes, MHD, ...simulations.

8.3.4 Java Development Kit JDK / JVM progress

Over the last three major releases of the Java Development Kit (JDK) the Java Virtual Machine (JVM) has delivered substantial performance progress both for the numerical and the IO performance. The JDK comes with two different JVMs, the Client VM and the Server VM. While the Client VM is optimized for fast response on User interaction and visualization, the Server VM is optimized for IO and computation. The results in this chapter are done with the Server VM only.

Computer system	Sun Fire X4600, AMD Opteron 8384, 64GB, 32 cores
OS	Solaris 10 05/09 s10x_u7 X86

8.3.4.1 Numeric performance

For the numerical performance results the EXTV grid with 780 Blocks was used, running the optimized ***JUSTEuler 3D*** solver with an additional load factor of 100.

Java Development Kit, Server VM	computing time 32 Bit	computing time 64 Bit
1.4.2_18	1570,516	
1.5.0_18	1403,052	1591,733
1.6.0_13	1340,101	1365,683

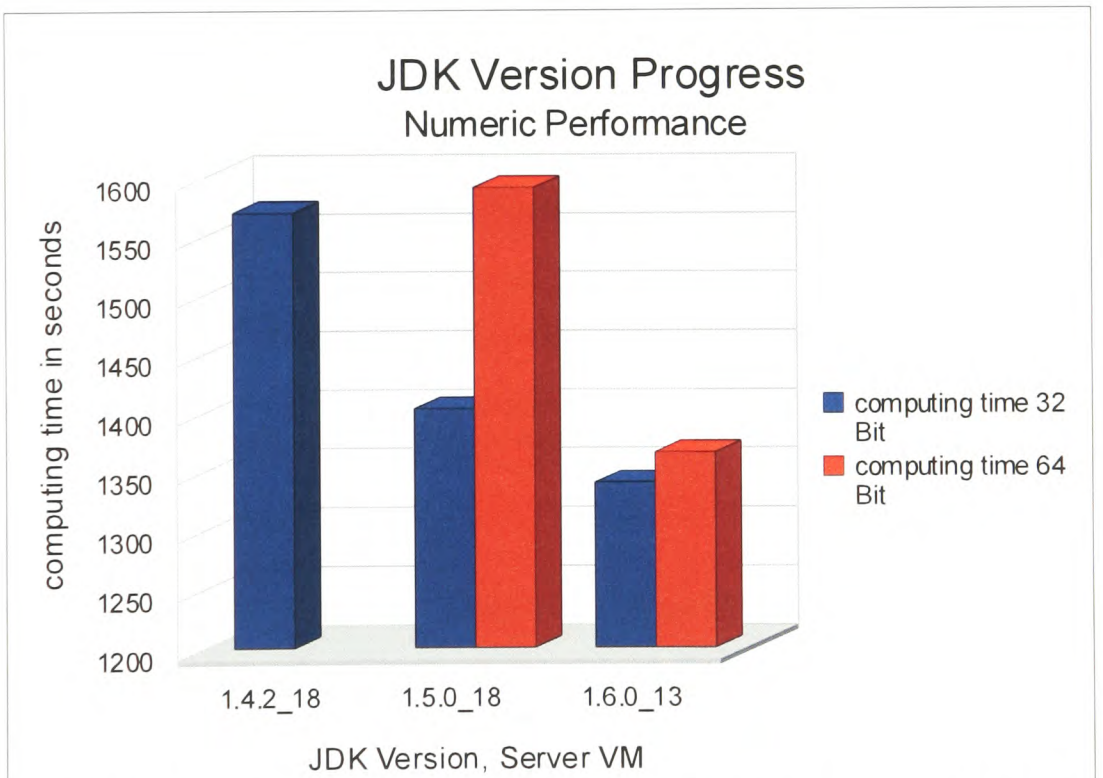


Illustration 8.3.10: The numeric performance progress of the last three major releases of the server Java Virtual Machine JVM

Since JDK 1.5 a 64 Bit Server VM is available for Solaris X86 systems. Because of the limitation to 4GB memory of the 32 Bit VM, the progress of the 64 Bit VM is of major interest. It is important to observe that most of the progress of the numeric performance was done within the 64 Bit Server VM.

8.3.4.2 IO Performance

As IO performance test the receiving times for the computed solution of the EXTV 780 Block grid were used.

Java Development Kit, Server VM	IO 32 Bit	IO 64 Bit
1.4.2_18	24,693	
1.5.0_18	22,594	14,987
1.6.0_13	16,651	10,440

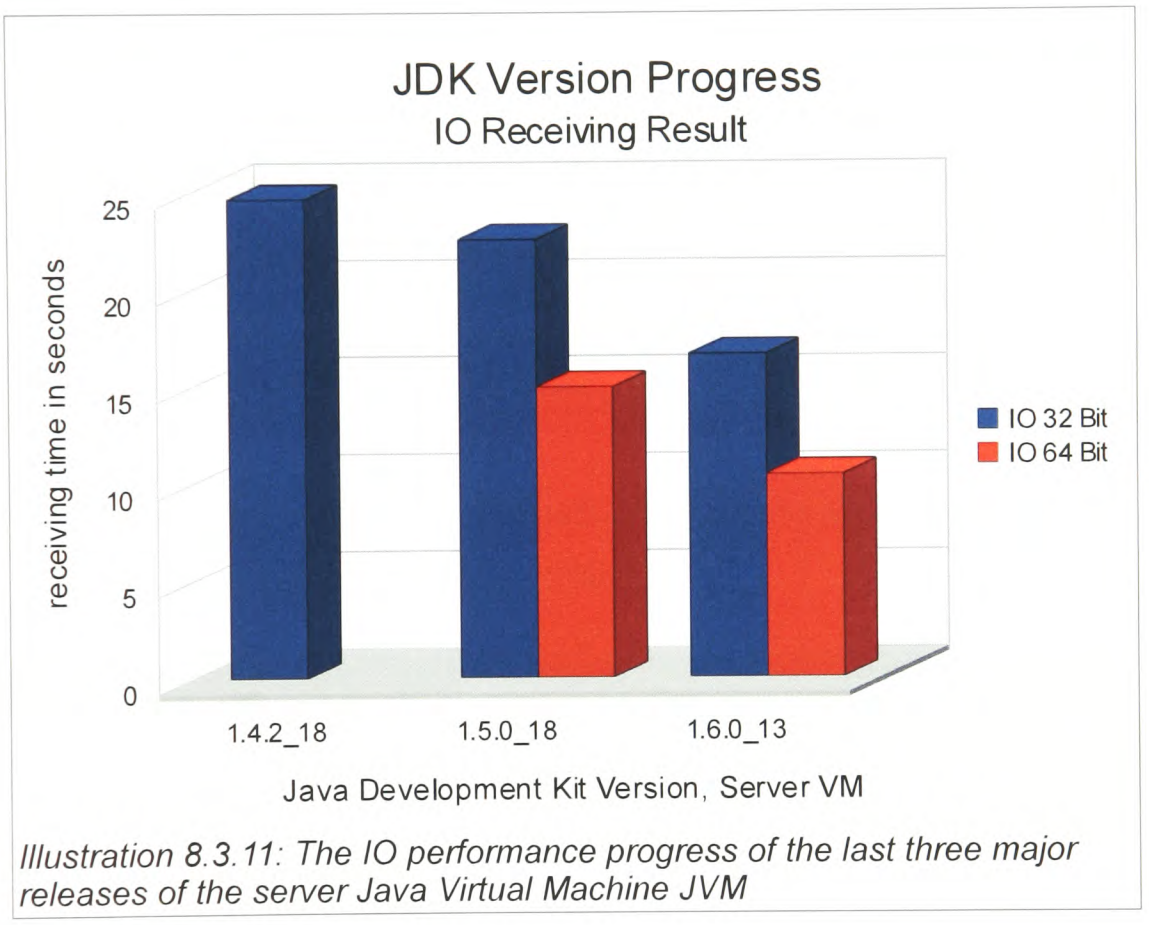


Illustration 8.3.11: The IO performance progress of the last three major releases of the server Java Virtual Machine JVM

While at this time the 32 Bit VM is slightly faster regarding the numeric performance than the 64 Bit VM (with the JDK 1.6), the IO performance of the 64 Bit Server VM was fbetter than the 32 Bit VM at all versions of the JDK. But it is encouraging to see that the IO performance is also progressing with every release of the JDK.

8.3.5 Operating System comparison

To examine the influence of the underlaying operating system exactly the same version of the Java Development Kit was run employing the same Hardware. The following Operating Systems where installed on the Sun Fire X4600:

- Linux CentOS 5.3, 64 Bit
- Microsoft, Windows Server 2008 HPC Edition, 64 Bit
- Sun Microsystems, Solaris 10 05/09, 64 Bit

Computer system	Sun Fire X4600, AMD Opteron 8384, 64GB, 32 cores
Java Development Kit (JDK)	JDK 1.6.0_13, 64Bit, Server VM, Parallel Garbage Collector
JDK parameter	java -d64 -server -Xcomp -Xnoclassgc -Xms4096m -Xmx4096m -XX:MaxPermSize=512m -XX: +UseFastAccessorMethods -XX:+UseParallelGC eulersolver3d.Main

Again, for these benchmarks the EXTV grid with 780 Blocks was used, running the optimized ***JUSTEuler 3D*** solver with an additional load of 100.

Illustration 8.3.12: A screen shot of Windows Server 2008s task manager and a linux perfbar binary.

On all three operations systems the utilization of the available 32 cores was 100%.

8.3.5.1 Timing, parallel efficiency and speedup results for the different operating systems

Linux CentOS 5.3, 64 Bit

#cores	time	efficiency	speedup
4	10516,928	100,00%	4,00
8	5258,885	99,99%	8,00
12	3552,032	98,69%	11,84
16	2624,060	100,20%	16,03
20	2099,458	100,19%	20,04
24	1761,249	99,52%	23,89
28	1538,817	97,63%	27,34
32	1333,300	98,60%	31,55

Receiving result = 11,531s

Windows Server 2008 HPC Edition, 64 Bit

#cores	time	efficiency	speedup
4	10476,461	100,00%	4,00
8	5301,133	98,81%	7,91
12	3555,450	98,22%	11,79
16	2641,238	99,16%	15,87
20	2119,352	98,86%	19,77
24	1760,596	99,18%	23,80
28	1514,504	98,82%	27,67
32	1332,108	98,31%	31,46

Receiving result = 13,249s

Solaris 10 05/09, 64 Bit Server VM

#cores	time	efficiency	speedup
4	10469,228	100,00%	4,00
8	5260,483	99,51%	7,96
12	3521,664	99,09%	11,89
16	2629,971	99,52%	15,92
20	2103,885	99,52%	19,90
24	1763,857	98,92%	23,74
28	1521,113	98,32%	27,53
32	1330,124	98,39%	31,48

Receiving result = 10.040s

There were no significant differences in the parallel efficiency and speedup between the three operating systems. The only timing differences are in the IO subsystem, which is fastest on Solaris.

Solaris has a real advantage against Linux and Windows running these tests due to the fact that Solaris is capable to switch cpus on and off while the OS is running. To change the number of available processors for Linux or Windows, a reboot of the OS is required. Especially under Windows an annoying strange behavior is that a restart only all of the available CPUs/cores can be utilized, and/or the number of available processors can be decreased only but not increased.

9 Conclusions and future work

With the advent of highly powerful parallel computers simulation science has become the third pillar of gaining knowledge and information besides classical experiment and mathematical analysis in understanding complex science and engineering problems. These problems are described by a multidisciplinary approach, and thus multiphysics formulation for complex geometries is the enabling technology of simulation science. Both the handling of complex geometry and the implementation of an efficient parallel strategy as well as the setup of a general numerical procedure are tasks that are common to most of the multiphysics problems but generally outside the expertise of the scientist or engineer who actually wishes to perform the simulation. Moreover, the implementation of the necessary algorithms for complex three dimensional geometries in combination with a general numerical solution procedure for a large class of physics problems demands special skills in computational geometry and computer science.

Since these tasks are common to a wide class of simulation problems the implementation of a versatile framework that provides these basic features seems to be most useful. Of course, to render such a framework useful for the simulation scientist a straight forward procedure for the integration of user defined multiphysics solvers must be provided. With ***JUSTGRID*** an "easy to integrate" simulation software framework is available for performing these tasks in an efficient and effective manner for high performance computation and communication using the Java™ runtime environment without any additional 3rd party libraries. ***JUSTGRID*** was created from scratch and contains more than 76,000 lines of code. ***JUSTGRID*** implements a new way of high performance computing and is streamlined for the new upcoming massive multi core processors that will dominate the computational scene within the next two years.

It can be used for all kinds of simulation, in particular for multiphysics problems described by systems of hyperbolic conservation laws (linear and nonlinear), based on the integral formulation of the conservation laws.

JUSTGRID framework takes care of all geometrical complexity, which is one of the most difficult parts in three dimensional simulations, and provides complete static as well as dynamic load balancing. Dynamic load balancing may be of crucial importance when a user needs to implement a numerical technique depending on its Krylov space dimension. For instance, if a shock wave is moving through a solution domain the Krylov space dimension is drastically increased at the location of the shock front, thus leading to a high numerical load that is also moving through the solution domain. As a result large spatial and temporal computational load imbalance may be generated. This kind of load imbalance can also be generated if the level of complexity of the physical model utilized is varying throughout the solution domain. To cope with these kinds of

problems dynamic load balancing needs to be employed. As a multithreaded application, ***JUSTGRID*** is able to run thousands of threads in a single process and achieves excellent dynamic load balancing. The various additional GUI-based Applications built around the ***JUSTGRID*** are assisting engineers during the complete simulation processes and providing testbeds for solver developers. Sample implementations of flow solvers (***JUSTSOLVER,*** Laplace, Euler, MHD Riemann) were created and successfully tested.

- For an existing grid it is now possible to provide 100% pure Java based applications for all parts of a simulation for systems of hyperbolic conservation laws, based on the integral form of the conservation equations.
- With ***JUST*** a modern, well structured, easy to use and extensible framework can be built. (***JUSTGRID***).
- Sample implementations of flow solvers (***JUSTSOLVER***) are available.
- Performance is on par with legacy 'C' code solver.
- ***JUSTGRID*** achieves a better speedup than ParNSS solver written in C.
- Removing all debugging code will further increase the performance of ***JUSTGRID***.

In the future all individual tools developed during this work will be merged into one workbench. The prepared but not implemented interface for cluster computing on distributed memory systems needs to be implemented. Performance analysis for different data exchange possibilities in distributed memory systems must also be done. Research on a better way of integrating legacy native code written in a different language should also be performed.

Appendices

Appendix Table of Contents

A File Formats

A.1 input

A.1.1 GRX

type: grx

GRX is our own defined format and is a validated XML file format storing additional information, namely: description *of what ?*, physical parameters and boundary conditions. The XML file with its corresponding DTD (Document Type Definition) together with the result of the computation is automatically stored as a ZIP-file into the user's file system with the file extension GRX. The ZIP-file format is a well known format available on any major computer system (UNIX/Linux, Windows, MacOS, etc.), and can be extracted with tools like Java's JAR, UNZIP or WinZIP.

A.1.2 Plot3D

type: p3d

Plot3D is a simple binary file format, used to represent structured curvilinear grids and scalar or vector fields defined on these grids. This format originates from the Plot3D program developed by Pieter Buning [PBU01] at NASA Ames.

A.1.3 GridPro Grid

type: gpg

Grid definition file in GridPro™ format. (see [GRP01])

A.1.4 GridPro Topology

type: gpc

Grid topology definition file containing the boundary conditions also. (see [GRP01])

A.1.5 ParNSS Command

type: cmd

ParNSS is our legacy Navier-Stokes solver written in C. The command file is an ASCII file containing information block connectivity, boundary conditions and the rotation between two connecting block faces. (see: [HAU01])

A.1.6 ParNSS Boundary

type:bnd

The ParNSS boundary file contains the boundary conditions only.

A.1.7 HGP XML

type: xml

Is a XML file containing grid and physical and numerical information in one file. It was designed by Dr. Hans-Georg Paap (HPCC Consultant, Barbing, Germany).

A.2 output

A.2.1 Tecplot

type: plt

Tecplot is a well known ASCII and binary format for storing CFD data. (see: [TPL01])

A.2.2 GRX

type: grx, see: A.1.1

A.2.3 GridPro Grid

type: gpg, see: A.1.3

A.2.4 Plot3D

type: p3d, see: A.1.2

A.2.5 ParNSS Command

type: cmd, see: A.1.5

A.2.6 ParNSS Boundary

type: bnd, see: A.1.6

B Java APIs used in JUSTGrid

Here is a list of Java API that are heavily used by ***JUST*** applications.

B.1 RMI

package: java.rmi

The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides for remote communication between programs written in the Java programming language.

B.2 Reflection API

package: java.lang.reflect

The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. With the reflection API one can:

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and super classes.
- Find out what constants and method declarations belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

B.3 Thread

class: java.lang.Thread

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

B.4 Java 2D

package: javax.swing

class: java.awt.Graphics2D

The Java 2D API is a set of classes for advanced 2D graphics and imaging, encompassing line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators.

B.5 Java 3D

package: javax.media.3d

URL: https://java3d.dev.java.net/

Java 3D is a scene graph-based 3D application programming interface (API) for the Java platform. It runs on top of either OpenGL or Direct3D.

B.6 Media Framework

package: javax.media.3d

URL: http://java.sun.com/products/java-media/jmf/

The Java Media Framework (JMF) is a Java Library that enables audio, video and other time-based media to be added to Java applications and applets. This optional package, which can capture, playback, stream, and transcode multiple media formats, extends the Java Platform, Standard Edition (Java SE) and allows development of cross-platform multimedia applications.

C JUSTSolver Template - Laplace 3D - Java API

The JavaDoc API documentation in this chapter is only is subset of the complete ***JUSTGrid*** framework API documentation. This subset contains all documentation needed by the ***JUSTSolver*** sources in the next chapter.

Package hpcc.just.domain

Interface JpBoundaryCondition

hpcc.just.domain

---

public interface **JpBoundaryCondition**

The JpBoundaryCondition condition interface simply define the NOT_SET variable.

Author:
Thorsten Ludewig

Field Summary	Page

String	**NOT_SET** Boundary condition NOT_SET	*158*

Field Detail

NOT_SET

```
public static final String NOT_SET
```

Boundary condition NOT_SET

Interface JpBoundaryHandler

hpcc.just.domain

All Superinterfaces:
Serializable

All Known Implementing Classes:
SimpleBoundaryHandler

---

public interface **JpBoundaryHandler**
extends Serializable

The JpBoundaryHandler interface.

Author:
Thorsten Ludewig

Method Summary		Page
void	**init**(JpBlock jpBlock) Initialization of the boundary handler	*158*
void	**setFaces**(JpBoundaryCondition[] faces) Sets the boundary condition array for the block faces	*159*
void	**updateBoundaryConditions**(int type) Before every single compute iteration this method will be executed by the JUSTGrid framework.	*159*

Method Detail

init

```
public void init(JpBlock jpBlock)
```

Initialization of the boundary handler

Parameters:

`jpBlock` - Reference to the parent block

---

setFaces

```
public void setFaces(JpBoundaryCondition[] faces)
```

Sets the boundary condition array for the block faces

Parameters:

`faces` - the faces array

---

updateBoundaryConditions

```
public void updateBoundaryConditions(int type)
```

Before every single compute iteration this method will be executed by the JUSTGrid framework.

Parameters:

`type` - The type of the boundary update.

Class JpCell

hpcc.just.domain

```
java.lang.Object
  └ hpcc.just.domain.JpCell
```

All Implemented Interfaces:

Cloneable, Serializable

Direct Known Subclasses:

SimpleCell2

---

abstract public class **JpCell**
extends Object
implements Serializable, Cloneable

This class represents a cell in the solution domain it contains information like face area, finite volume, ... but no U vector!

Author:

Thorsten Ludewig

Field Summary	Page

double	**finiteVolume** The finite volume	*160*
static final int	**NUMBER_OF_FACES** The number of cell faces - since we are using cubes: NUMBER_OF_FACES = 6	*160*

Constructor Summary	Page
JpCell()	*161*

Method Summary		Page
abstract Object	**getData**() This method must return an object containing all boundary exchange information for this cell.	*161*
JpVector	**getFaceVector**(int face) Gets a face vector for one cell face **ATTENTION!**	*161*
JpVector[]	**getFaceVectors**() Gets an array containing all face vectors for this cell	*161*
double	**getFiniteVolume**() Gets the finite volume of this cell	*161*
abstract void	**setData**(Object data) This method sets all infomation from a baoundary exchange.	*162*
void	**setFaceVector**(int face, JpVector faceVector) Sets a face vector for one cell face **ATTENTION!**	*162*
void	**setFiniteVolume**(double finiteVolume) Sets the finite volume of this cell	*162*

Field Detail

NUMBER_OF_FACES

```
public static final int NUMBER_OF_FACES
```

The number of cell faces - since we are using cubes: NUMBER_OF_FACES = 6

---

finiteVolume

```
public double finiteVolume
```

The finite volume

Constructor Detail

JpCell

```
public JpCell()
```

Method Detail

getData

```
public abstract Object getData()
```

This method must return an object containing all boundary exchange information for this cell.

Returns:

the exchange information

---

getFaceVector

```
public JpVector getFaceVector(int face)
```

Gets a face vector for one cell face

ATTENTION! The index starts at 1 *not* at 0

Parameters:

`face` - the face index (from 1 to 6)

Returns:

the face vector for the specified face

See Also:

`getFaceVectors()`, `setFaceVector( int face, JpVector faceVector )`, `NUMBER_OF_FACES`

---

getFaceVectors

```
public JpVector[] getFaceVectors()
```

Gets an array containing all face vectors for this cell

Returns:

this cell's face vectors

See Also:

`getFaceVector( int face )`, `setFaceVector( int face, JpVector faceVector )`, `NUMBER_OF_FACES`

---

getFiniteVolume

```
public double getFiniteVolume()
```

Gets the finite volume of this cell

Returns:

this cell's finite volume

See Also:

`setFiniteVolume( double finiteVolume )`

---

setData

```
public abstract void setData(Object data)
```

This method sets all information from a boundary exchange.

Parameters:

`data` - the neigboring cell information

---

setFaceVector

```
public void setFaceVector(int face,
                          JpVector faceVector)
```

Sets a face vector for one cell face

ATTENTION! The index starts at 1 *not* at 0

Parameters:

`face` - the face index (from 1 to 6)

`faceVector` - the face normal vector the length represents the area of the cell face

See Also:

`getFaceVectors()`, `setFaceVector( int face, JpVector faceVector )`, `NUMBER_OF_FACES`

---

setFiniteVolume

```
public void setFiniteVolume(double finiteVolume)
```

Sets the finite volume of this cell

Parameters:

`finiteVolume` - the finite volume

See Also:

`getFiniteVolume()`

Interface JpDomain

hpcc.just.domain

All Known Implementing Classes:

JpBlock

---

public interface **JpDomain**

This interface lists the requirements that a domain must meet. A domain contains cells that holds the variable to be computed

Author:

Thorsten Ludewig

Method Summary		Page
JpCell	**getJpCell**(int i) this method returns the cell at index i	163
JpCell	**getNeighbor**(int face, int depth, JpCell cell) this method returns a neighboring cell specified by the face, the depth (i.e. area of influence of the numerical scheme) and the original cell	163
int	**getNumberOfCells**() this method returns the number of cells that resides in the domain	164
void	**updateBoundaryConditions**(int type) This method will be executed by the framework to initiate the update of the boundary conditions.	164

Method Detail

getJpCell

```
public JpCell getJpCell(int i)
```

this method returns the cell at index i

Parameters:

`i` - index of a cell in the domain

Returns:

cell object of type JpCell

---

getNeighbor

```
public JpCell getNeighbor(int face,
                          int depth,
                          JpCell cell)
```

this method returns a neighbouring cell specified by the face, the depth (i.e. area of

influence of the numerical scheme) and the original cell

Parameters:

`face` - face of the cell

`depth` - depth from the face

`cell` - the original cell

Returns:

neighbouring cell of type JpCell

---

getNumberOfCells

```
public int getNumberOfCells()
```

this method returns the number of cells that resides in the domain

Returns:

number of cells; of type int

---

updateBoundaryConditions

```
public void updateBoundaryConditions(int type)
```

This method will be executed by the framework to initiate the update of the boundary conditions.

Parameters:

`type` - the type of the boundary condition update (not necessary)

Class JpFace

hpcc.just.domain

```
java.lang.Object
  └ hpcc.just.domain.JpFace
```

---

public class **JpFace**
extends Object

JpFace contains all information for a structured block face.

Author:

Thorsten Ludewig

Field Summary		Page
static final int	**I_MAX** I_MAX = 4	166

static final int	**I_MIN** I_MIN = 3	*166*
static final int	**J_MAX** J_MAX = 5	*166*
static final int	**J_MIN** J_MIN = 2	*166*
static final int	**K_MAX** K_MAX = 6	*165*
static final int	**K_MIN** K_MIN = 1	*165*

Constructor Summary	Page
JpFace(int faceNumber, JpBlock parent) Constructor of a block face	*166*

Method Summary		Page
int	**getFaceNumber**() Gets the block face namuber	*166*
JpFacePart	**getFacePart**(int facePartNumber) JUSTGrid is prepared for merged grid, which means a bock face can have multiple face parts connecting to other face parts on different blocks.	*166*
JpFacePart[]	**getFaceParts**() Return all face parts	*167*
JpBlock	**getParentBlock**() Gets the parent block	*167*
void	**setFaceNumber**(int faceNumber) Sets the unique face number	*167*
void	**setFacePart**(JpFacePart facePart, int facePartNumber) Sets a reference to a face part	*167*

Field Detail

K_MIN

```
public static final int K_MIN
```

K_MIN = 1

K_MAX

```
public static final int K_MAX
```

K_MAX = 6

J_MIN

```
public static final int J_MIN
```

J_MIN = 2

J_MAX

```
public static final int J_MAX
```

J_MAX = 5

I_MIN

```
public static final int I_MIN
```

I_MIN = 3

I_MAX

```
public static final int I_MAX
```

I_MAX = 4

Constructor Detail

JpFace

```
public JpFace(int faceNumber,
              JpBlock parent)
```

Constructor of a block face

Method Detail

getFaceNumber

```
public int getFaceNumber()
```

Gets the block face number

Returns:

the face number

---

getFacePart

```
public JpFacePart getFacePart(int facePartNumber)
```

JUSTGrid is prepared for merged grid, which means a bock face can have multiple face parts connecting to other face parts on different blocks. The implementation at this time is for 1 face part only.

Parameters:

`facePartNumber` - the face part number

Returns:

the reference to the face part

---

getFaceParts

```
public JpFacePart[] getFaceParts()
```

Return all face parts

Returns:

an array containing all face parts

---

getParentBlock

```
public JpBlock getParentBlock()
```

Gets the parent block

Returns:

the reference to the parent block

---

setFaceNumber

```
public void setFaceNumber(int faceNumber)
```

Sets the unique face number

Parameters:

`faceNumber` - the unique face number 1 ... 6

---

setFacePart

```
public void setFacePart(JpFacePart facePart,
                        int facePartNumber)
```

Sets a reference to a face part

Parameters:

`facePart` - the reference to the face part

`facePartNumber` - the unique face part number - at this time always 1

Class JpFacePart

hpcc.just.domain

```
java.lang.Object
  └ hpcc.just.domain.JpFacePart
```

---

public class **JpFacePart**
extends Object

This class represents an area on a block face. At this time there is only one face part per block face.

Author:
Thorsten Ludewig
Version:
1.0
See Also:
JpFace

Constructor Summary	Page
JpFacePart(int partNumber, JpFace parent) Constructor declaration	170

Method Summary		Page
String	**getBoundaryCondition**() Get the boundary condition of the face part	170
int	**getIteration**() The current iteration computed on this face part	170
int	**getNeighborBlockNumber**() Getting the block number of the neighbouring block	170
int	**getNeighborFaceNumber**() Get the number of the neighbouring face	170
JpFacePart	**getNeighborFacePart**() Get a reference to the neighbouring face part	171
int	**getNeighborFacePartNumber**() Getting the part number of the neighbouring face part	171
int	**getOrientation**() Getting the orientation to the neighbouring face part	171
JpFace	**getParentFace**() Getting the reference to the parent face	171
int	**getPartHeight**() Getting the height of the part face	171

int	**getPartNumber**() Getting the part number	171
int	**getPartWidth**() Getting the part width	172
int	**getPartX**() The x position on the block face of the face part.	172
int	**getPartY**() The y position on the block face of the face part.	172
void	**init**() Initialize the face part	172
void	**nextIteration**() Increase the current iteration to the next iteration	172
Object[][][]	**readCommunicationBuffer**() Reading the communication buffer of the face part with respect to the orientation to the neighbour face part.	172
void	**setBoundaryCondition**(String boundaryCondition) Sets the boundary condition of the face part	173
void	**setNeighborBlockNumber**(int neighborBlockNumber) Setting the block number of the neighbouring block	173
void	**setNeighborFaceNumber**(int neighborFaceNumber) Setting the part number of the neighbouring face	173
void	**setNeighborFacePart**(JpFacePart neighborFacePart) Sets a reference to the neighbouring face part	173
void	**setNeighborFacePartNumber**(int neighborFacePartNumber) Setting the part number of the neighbouring face part	173
void	**setOrientation**(int orientation) Setting the orientation to the neighbouring face part	174
void	**setPartHeight**(int partHeight) Setting the height of the part face	174
void	**setPartNumber**(int partNumber) Setting the unique face part number	174
void	**setPartWidth**(int partWidth) Setting the width of the part face	174
void	**setPartX**(int partX) The x position on the block face of the face part.	174
void	**setPartY**(int partY) The y position on the block face of the face part.	174
void	**writeCommunicationBuffer**(Object[][][] buffer) Writing the communication buffer to the halo cells.	175

Constructor Detail

JpFacePart

```
public JpFacePart(int partNumber,
                  JpFace parent)
```

Constructor declaration

Method Detail

getBoundaryCondition

```
public String getBoundaryCondition()
```

Get the boundary condition of the face part

Returns:

A string representing the boundary condition

---

getIteration

```
public int getIteration()
```

The current iteration computed on this face part

Returns:

the iteration

---

getNeighborBlockNumber

```
public int getNeighborBlockNumber()
```

Getting the block number of the neighbouring block

Returns:

the neighbour block number

---

getNeighborFaceNumber

```
public int getNeighborFaceNumber()
```

Get the number of the neighbouring face

Returns:

the neighbouring face number

---

getNeighborFacePart

```
public JpFacePart getNeighborFacePart()
```

Get a reference to the neighbouring face part
Returns:
the reference to the neighbouring face part

---

getNeighborFacePartNumber

```
public int getNeighborFacePartNumber()
```

Getting the part number of the neighbouring face part
Returns:
the part number of the neighbouring face part

---

getOrientation

```
public int getOrientation()
```

Getting the orientation to the neighbouring face part
Returns:
the orientation

---

getParentFace

```
public JpFace getParentFace()
```

Getting the reference to the parent face
Returns:
the reference to the parent face

---

getPartHeight

```
public int getPartHeight()
```

Getting the height of the part face
Returns:
the height

---

getPartNumber

```
public int getPartNumber()
```

Getting the part number
Returns:
the current part number

---

getPartWidth

```
public int getPartWidth()
```

Getting the part width
Returns:
the width of the part

---

getPartX

```
public int getPartX()
```

The x position on the block face of the face part. At this time always 0
Returns:
the x position

---

getPartY

```
public int getPartY()
```

The y position on the block face of the face part. At this time always 0
Returns:
the y position

---

init

```
public void init()
```

Initialize the face part

---

nextIteration

```
public void nextIteration()
```

Increase the current iteration to the next iteration

---

readCommunicationBuffer

```
public Object[][][] readCommunicationBuffer()
```

Reading the communication buffer of the face part with respect to the orientation to the neighbour face part. (Boundary exchange)
Returns:
An array containing all data for the boundary exchange

---

setBoundaryCondition

```
public void setBoundaryCondition(String boundaryCondition)
```

Sets the boundary condition of the face part
Parameters:
`boundaryCondition` - A string representing the boundary condition

---

setNeighborBlockNumber

```
public void setNeighborBlockNumber(int neighborBlockNumber)
```

Setting the block number of the neighbouring block
Parameters:
`neighborBlockNumber` - the block number of the neighbouring block

---

setNeighborFaceNumber

```
public void setNeighborFaceNumber(int neighborFaceNumber)
```

Setting the part number of the neighbouring face
Parameters:
`neighborFaceNumber` - the number of the neighbouring face

---

setNeighborFacePart

```
public void setNeighborFacePart(JpFacePart neighborFacePart)
```

Sets a reference to the neigboring face part
Parameters:
`neighborFacePart` - the reference to the neighbouring face part

---

setNeighborFacePartNumber

```
public void setNeighborFacePartNumber(int neighborFacePartNumber)
```

Setting the part number of the neighboring face part
Parameters:
`neighborFacePartNumber` - the part number of the neighbouring face part

---

setOrientation

```
public void setOrientation(int orientation)
```

Setting the orientation to the neighbouring face part
Parameters:

`orientation` - the orientation 1 ... 8

---

setPartHeight

```
public void setPartHeight(int partHeight)
```

Setting the height of the part face
Parameters:

`partHeight` - the height of the part face

---

setPartNumber

```
public void setPartNumber(int partNumber)
```

Setting the unique face part number
Parameters:

`partNumber` - the unique face part number

---

setPartWidth

```
public void setPartWidth(int partWidth)
```

Setting the width of the part face
Parameters:

`partWidth` - the width of the part face

---

setPartX

```
public void setPartX(int partX)
```

The x position on the block face of the face part. At this time it should be always 0
Parameters:

`partX` - The x position

---

setPartY

```
public void setPartY(int partY)
```

The y position on the block face of the face part. At this time it should be always 0

Parameters:

`partY` - The y position

---

writeCommunicationBuffer

```
public void writeCommunicationBuffer(Object[][][] buffer)
```

Writing the communication buffer to the halo cells.

Parameters

`buffer` - the data array

Package hpcc.just.domain.structured

Class Summary		Page
JpBlock	A JpBlock is the representation for a single strucktured domain (block).	175

Class JpBlock

hpcc.just.domain.structured

```
java.lang.Object
  └ hpcc.just.domain.structured.JpBlock
```

All Implemented Interfaces:

JpDomain

---

public class **JpBlock**
extends Object
implements JpDomain

A JpBlock is the representation for a single structured domain (block). It it is the parent container for: solver, cells, geometry and boundary handler.

Author:

Thorsten Ludewig

Field Summary		Page
static final int	**NUMBER_OF_FACES** In this implementation the number of faces is fixed to 6.	177

Constructor Summary	Page

JpBlock(int uniqueId, int gridI, int gridJ, int gridK) Constructor declaration	*178*

Method Summary		Page
JpBlock[]	**getBlockArray**() Returns the reference to the all blocks array	*178*
String	**getBlockName**() Returns the free defined block name	*178*
int	**getBlockNumber**() Returns the block number	*178*
JpBoundaryHandler	**getBoundaryHandler**() Returns the associaded boundary handler object.	*178*
JpCell	**getCell**(int i, int j) Returns one cell in 2D.	*179*
JpCell	**getCell**(int i, int j, int k) getJpCell returns the [i,j,k] given cell **ATTENTION!**	*179*
JpCell[][][]	**getCells**() JpCell returns a reference to the 3D JpCell array	*179*
JpFace	**getFace**(int faceNo) Returns a block face object	*179*
JpFace[]	**getFaces**() Returns an array containing all block faces	*180*
int	**getGridI**() Returns the number of grid points in I direction.	*180*
int	**getGridJ**() Returns the number of grid points in J direction.	*180*
int	**getGridK**() Returns the number of grid points in K direction.	*180*
JpVector	**getGridVector**(int i, int j, int k) Returns a single grid point.	*180*
JpVector[][][]	**getGridVectorArray**() The complete grid for this block.	*181*
int	**getGridVectorCount**() Returns the total number of grid points	*181*
JpCell	**getJpCell**(int i) **Deprecated.** *this method is obsolete*	*181*
JpCell	**getNeighbor**(int face, int depth, JpCell cell) **Deprecated.** *this method is obsolete*	*181*
int	**getNumberOfCells**() **Deprecated.** *this method is obsolete*	*182*
int	**getNumberOfHaloCells**() Returns the specified number of halo cells	*182*

JpSolver	**getSolver**() Returns a reference to the solver object	182
int	**getUniqueId**() Returns the unique block id	182
void	**setBlockArray**(JpBlock[] blockArray) Set the reference to the all block array	183
void	**setBlockName**(String blockName) Sets the block name	183
void	**setBlockNumber**(int blockNumber) Sets the block number	183
void	**setBoundaryHandler**(JpBoundaryHandler boundaryHandler) Sets the boundary handler object.	183
void	**setCells**(JpCell jpCell, int numberOfHaloCells) This method initialize the complete cell array including the halo cells	183
void	**setFace**(JpFace face, int faceNo) Sets the block face object.	184
void	**setGridI**(int gridI) Sets the number of grid points in I direction	184
void	**setGridJ**(int gridJ) Sets the number of grid points in J direction	184
void	**setGridK**(int gridK) Sets the number of grid points in K direction	184
void	**setGridVector**(JpVector vector, int i, int j, int k) Sets a single grid point vector	184
void	**setSolver**(JpSolver solver) Sets the solver object.	185
void	**updateBoundaryConditions**(int type) Calls the update boundary	185

Methods inherited from interface hpcc.just.domain.JpDomain
getJpCell, getNeighbor, getNumberOfCells, updateBoundaryConditions

Field Detail

NUMBER_OF_FACES

```
public static final int NUMBER_OF_FACES
```

In this implementation the number of faces is fixed to 6.

Constructor Detail

JpBlock

```
public JpBlock(int uniqueId,
               int gridI,
               int gridJ,
               int gridK)
```

Constructor declaration

Method Detail

getBlockArray

```
public JpBlock[] getBlockArray()
```

Returns the reference to the all blocks array
Returns:
the array containing all blocks

---

getBlockName

```
public String getBlockName()
```

Returns the free defined block name
Returns:
te block name

---

getBlockNumber

```
public int getBlockNumber()
```

Returns the block number
Returns:
the block number

---

getBoundaryHandler

```
public JpBoundaryHandler getBoundaryHandler()
```

Returns the associated boundary handler object.
Returns:
the boundary handler

---

getCell

```
public JpCell getCell(int i,
                      int j)
```

Returns one cell in 2D. The index starts at 1!

Parameters:

`i` - i index

`j` - j index

Returns:

the cell object

---

getCell

```
public JpCell getCell(int i,
                      int j,
                      int k)
```

getJpCell returns the [i,j,k] given cell

ATTENTION! the start index of i,j,k is 1 not 0 and ends at (e.g for i) gridI - 1 so your loop should look like this:

```
int gridI = jpBlock.getgridI();
for ( int i=1; i
```

The number of cell in each direction is the number of grid points in that direction - 1!

Parameters:

`i` - cell index in i direction

`j` - cell index in j direction

`k` - cell index in k direction

Returns:

returns the [i,j,k] given cell

---

getCells

```
public JpCell[][][] getCells()
```

JpCell returns a reference to the 3D JpCell array

Returns:

the reference to the 3D JpCell array

See Also:

`getCell(int i, int j, int k )`

---

getFace

```
public JpFace getFace(int faceNo)
```

Returns a block face object
Parameters:
`faceNo` - face number 1 ... 6
Returns:
the JpFace

---

getFaces

```
public JpFace[] getFaces()
```

Returns an array containing all block faces
Returns:
the array

---

getGridI

```
public int getGridI()
```

Returns the number of grid points in I direction.
Returns:
number of grid points in I direction.

---

getGridJ

```
public int getGridJ()
```

Returns the number of grid points in J direction.
Returns:
number of grid points in J direction.

---

getGridK

```
public int getGridK()
```

Returns the number of grid points in K direction.
Returns:
number of grid points in K direction.

---

getGridVector

```
public JpVector getGridVector(int i,
                              int j,
                              int k)
```

Returns a single grid point.

Parameters:

`i` - I

`j` - J

`k` - K

Returns:

the grid point

---

getGridVectorArray

```
public JpVector[][][] getGridVectorArray()
```

The complete grid for this block.

Returns:

a vector array with all grid points for this bock

---

getGridVectorCount

```
public int getGridVectorCount()
```

Returns the total number of grid points

Returns:

the total number of grid points

---

getJpCell

```
public JpCell getJpCell(int i)
```

Deprecated. *this method is obsolete*

Dummy method

Specified by:

`getJpCell` in interface `JpDomain`

Parameters:

`i` - type

Returns:

null

---

getNeighbor

```
public JpCell getNeighbor(int face,
                          int depth,
                          JpCell cell)
```

Deprecated. *this method is obsolete*

Dummy method

Specified by:

getNeighbor in interface JpDomain

Parameters:

face - Face

depth - Depth

cell - Cell

Returns:

null

---

getNumberOfCells

```
public int getNumberOfCells()
```

Deprecated. *this method is obsolete*

Dummy method

Specified by:

getNumberOfCells in interface JpDomain

Returns:

0

---

getNumberOfHaloCells

```
public int getNumberOfHaloCells()
```

Returns the specified number of halo cells

Returns:

the number of halo cells

---

getSolver

```
public JpSolver getSolver()
```

Returns a reference to the solver object

Returns:

the solver object

---

getUniqueId

```
public int getUniqueId()
```

Returns the unique block id

Returns:

the unique block id

---

setBlockArray

```
public void setBlockArray(JpBlock[] blockArray)
```

Set the reference to the all block array
Parameters:
`blockArray` - the all block array

---

setBlockName

```
public void setBlockName(String blockName)
```

Sets the block name
Parameters:
`blockName` - the block name

---

setBlockNumber

```
public void setBlockNumber(int blockNumber)
```

Sets the block number
Parameters:
`blockNumber` - the block number

---

setBoundaryHandler

```
public void setBoundaryHandler(JpBoundaryHandler boundaryHandler)
```

Sets the boundary handler object.
Parameters:
`boundaryHandler` - the boundary handler

---

setCells

```
public void setCells(JpCell jpCell,
                     int numberOfHaloCells)
```

This method initialize the complete cell array including the halo cells
Parameters:
`jpCell` - the prototype cell from that all cells be generated
`numberOfHaloCells` - number of halo cells

---

setFace

```
public void setFace(JpFace face,
                    int faceNo)
```

Sets the block face object.
Parameters:
`face` - the block face object
`faceNo` - the face number 1 ... 6

---

setGridI

```
public void setGridI(int gridI)
```

Sets the number of grid points in I direction
Parameters:
`gridI` - the number of grid points in I direction

---

setGridJ

```
public void setGridJ(int gridJ)
```

Sets the number of grid points in J direction
Parameters:
`gridJ` - the number of grid points in J direction

---

setGridK

```
public void setGridK(int gridK)
```

Sets the number of grid points in K direction
Parameters:
`gridK` - the number of grid points in K direction

---

setGridVector

```
public void setGridVector(JpVector vector,
                          int i,
                          int j,
                          int k)
```

Sets a single grid point vector
Parameters:
`vector` - the grid point
`i` - I
`j` - J

k - K

---

setSolver

```
public void setSolver(JpSolver solver)
```

Sets the solver object.

Parameters:

solver - the solver object

---

updateBoundaryConditions

```
public void updateBoundaryConditions(int type)
```

Calls the update boundary

Specified by:

updateBoundaryConditions in interface JpDomain

Parameters:

type - type of the boundary update

Package hpcc.just.share

Interface Summary		Page
JploStreamStatus	JParNSS io stream status interface	185
JpMultiblockSolver	Interface description for a multi block solver	187
JpServerSession	JParNSS Server Session interface is used for server side access to the JpSession class.	188
JpSolver	JParNSS Solver interface A client application must implement this interface.	191
JpSolverHandler	JpSolverHandler interface represents	195

Exception Summary		Page
JpSolverException	The JpSolverException will be thrown if an unexpected error will occur.	193

Interface JploStreamStatus

hpcc.just.share

---

public interface **JploStreamStatus**

JUSTGrid io stream status interface

Author:
Thorsten Ludewig

Method Summary		Page
void	**destroy** ( ) destroy the io stream	186
int	**getId** ( ) get the command id	186
void	**receiveAcknowledge** ( ) wait for receiving an acknowledge signal	186
void	**sendAcknowledge** ( ) send and acknowledge signal	186

Method Detail

destroy

```
public void destroy()
```

destroy the io stream

---

getId

```
public int getId()
```

get the command id
Returns:
the unique id of the io stream

---

receiveAcknowledge

```
public void receiveAcknowledge()
```

wait for receiving an acknowledge signal

---

sendAcknowledge

```
public void sendAcknowledge()
```

send and acknowledge signal

Interface JpMultiblockSolver

hpcc.just.share

All Superinterfaces:

JpSolver

All Known Implementing Classes:

LaplaceSolver3D

---

public interface **JpMultiblockSolver**
extends JpSolver

Interface description for a multi block solver

Author:

Thorsten Ludewig

Method Summary	Page
void **finalizeSolver**() This method finalizes the solver object on the server side.	187
void **initSolver**(JpDomain block, int nodeId) This method initializes the solver object on the server side.	187

Methods inherited from interface hpcc.just.share.JpSolver
getDataObject, getFaces, getOutputVars, setDataObject, solve

Method Detail

finalizeSolver

```
public void finalizeSolver()
```

This method finalizes the solver object on the server side.

---

initSolver

```
public void initSolver(JpDomain block,
                       int nodeId)
```

This method initializes the solver object on the server side.

Parameters:

`block` - the block to work on

`nodeId` - a unique node/solver id

Interface JpServerSession

hpcc.just.share

---

public interface **JpServerSession**

JUSTGrid Server Session interface is used for server side access to the JpSession class. The JpSolverHandler uses this interface.

Author:
Thorsten Ludewig

Method Summary		Page
JpSolver[]	**getSolverArray**() getting the solver array	189
JpSolverHandler	**getSolverHandler**() getting the current solver handler	189
void	**initAllServerBoundaryHandlers**(JpBoundaryHandler boundaryHandler) initializing all boundary handler on the server	189
void	**initAllServerCells**(JpCell cell, int haloCells) initialize all JpCells	189
void	**initAllServerNodes**(JpSolver solver) initialize all nodes	189
void	**initServerBoundaryHandler**(int blockIndex, JpBoundaryHandler boundaryHandler) Initialize one boundary handler	190
void	**initServerNode**(int nodeIndex) initialize a specific node	190
void	**initServerNode**(int nodeIndex, JpSolver solver) initialize a specific node with a JpSolver	190
void	**initServerSession**(JpBlock[] block) initialize the JpSession	190
void	**initServerSession**(int numberOfNodes, int maxNumberOfNeighbors) initialize the JpSession	190
void	**setServerNodeNeighborObject**(int nodeIndex, int neighborIndex, int edge) binding a nodes edge to a neighbour node (topology information)	191
void	**setSolverArray**(JpSolver[] solverArray) setting up a reference to the solver array	191

Method Detail

getSolverArray

```
public JpSolver[] getSolverArray()
```

getting the solver array

Returns:

a reference to the solver array

---

getSolverHandler

```
public JpSolverHandler getSolverHandler()
```

getting the current solver handler

Returns:

the current solver handler

---

initAllServerBoundaryHandlers

```
public void initAllServerBoundaryHandlers(JpBoundaryHandler boundaryHandler)
```

initializing all boundary handler on the server

Parameters:

`boundaryHandler` - a reference to a boundary handler object

---

initAllServerCells

```
public void initAllServerCells(JpCell cell,
                               int haloCells)
```

initialize all JpCells

Parameters:

`cell` - a reference to a cell object

`haloCells` - number of halo cells

---

initAllServerNodes

```
public void initAllServerNodes(JpSolver solver)
```

initialize all nodes

Parameters:

`solver` - a reference to a solver object

---

initServerBoundaryHandler

```
public void initServerBoundaryHandler(int blockIndex,
                                       JpBoundaryHandler boundaryHandler)
```

Initialize one boundary handler
Parameters:
`blockIndex` - index of the block
`boundaryHandler` - the reference to the boundary handler object

---

initServerNode

```
public void initServerNode(int nodeIndex)
```

initialize a specific node
Parameters:
`nodeIndex` - index of a node

---

initServerNode

```
public void initServerNode(int nodeIndex,
                           JpSolver solver)
```

initialize a specific node with a JpSolver
Parameters:
`nodeIndex` - index of a node / block
`solver` - a solver object

---

initServerSession

```
public void initServerSession(JpBlock[] block)
```

initialize the JpSession
Parameters:
`block` - the Multiblock structure

---

initServerSession

```
public void initServerSession(int numberOfNodes,
                              int maxNumberOfNeighbors)
```

initialize the JpSession
Parameters:
`numberOfNodes` - the total number of nodes/solvers with this session
`maxNumberOfNeighbors` - the maximum number of neighbour nodes for one node

setServerNodeNeighborObject

```
public void setServerNodeNeighborObject(int nodeIndex,
                                         int neighborIndex,
                                         int edge)
```

binding a nodes edge to a neighbour node (topology information)

Parameters:

`nodeIndex` - index of a node

`neighborIndex` - index of the neighbor node

`edge` - edge to bind with the neighbor node

---

setSolverArray

```
public void setSolverArray(JpSolver[] solverArray)
```

setting up a reference to the solver array

Parameters:

`solverArray` - a solver array

Interface JpSolver

hpcc.just.share

All Known Subinterfaces:

JpMultiblockSolver

All Known Implementing Classes:

LaplaceSolver3D

---

public interface **JpSolver**

JUSTGrid Solver interface A client application must implement this interface. Every server node makes a reference to one solver object.

Author:

Thorsten Ludewig

Method Summary		Page
Object	**getDataObject**(int dataId) get solver data from the solver object	192
JpBoundaryCondition[]	**getFaces**() Returns the faces array	192

Object	**getOutputVars**(int gridI, int gridJ, int gridK) Returns an object representing the flow vars for on grid point	192
void	**setDataObject**(int dataId, Object object) send data objects to the solver object	192
boolean	**solve**(int iteration) The ‚solve' method contains the numerics for ONE iteration.	193

Method Detail

getDataObject

```
public Object getDataObject(int dataId)
```

get solver data from the solver object

Parameters:

`dataId` - this parameter is used to select a specific object

Returns:

a data object

---

getFaces

```
public JpBoundaryCondition[] getFaces()
```

Returns the faces array

Returns:

the faces array

---

getOutputVars

```
public Object getOutputVars(int gridI,
                            int gridJ,
                            int gridK)
```

Returns an object representing the flow vars for on grid point

Parameters:

`gridI` - I

`gridJ` - J

`gridK` - K

Returns:

an object representing the flow vars for on grid point

---

setDataObject

```
public void setDataObject(int dataId,
                          Object object)
```

send data objects to the solver object

Parameters:

`dataId` - this parameter is used to select a specific object

`object` - the data object

---

solve

```
public boolean solve(int iteration)
             throws JpSolverException
```

The ‚solve' method contains the numerics for ONE iteration.

Parameters:

`iteration` - the current iteration

Returns:

is NOT ready

Throws:

`JpSolverException` - a user specific exception

Class JpSolverException

hpcc.just.share

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ hpcc.just.share.JpSolverException
```

All Implemented Interfaces:

Serializable

---

public class **JpSolverException**
extends Exception

The JpSolverException will be thrown if an unexpected error will occur.

Author:

Thorsten Ludewig

Constructor Summary	Page
JpSolverException() The default constructor	194
JpSolverException(String message) Constructor with a message	194

Method Summary		Page
String	**getMessage**() Returns the exception message	194
String	**toString**() Returns a string represents the JpSolverException	194

Constructor Detail

JpSolverException

```
public JpSolverException()
```

The default constructor

JpSolverException

```
public JpSolverException(String message)
```

Constructor with a message

Method Detail

getMessage

```
public String getMessage()
```

Returns the exception message

Overrides:

`getMessage` in class `Throwable`

Returns:

the exception message

---

toString

```
public String toString()
```

Returns a string represents the JpSolverException

Overrides:

`toString` in class `Throwable`

Returns:

a string represents the JpSolverException

Interface JpSolverHandler

hpcc.just.share

---

public interface **JpSolverHandler**

JpSolverHandler interface represents

Author:
Thorsten Ludewig

Method Summary		Page
void	**destroyHandler**() Destroy the session handler	195
void	**initHandler**(JpServerSession jpServerSession) Initialize the solver handler	195
void	**readData**(InputStream inputStream, OutputStream outputStream, JpIoStreamStatus jpIoStreamStatus, String command) Read data from session	196
void	**startSession**() Called from jpSession.startSession();	196
void	**writeData**(InputStream inputStream, OutputStream outputStream, JpIoStreamStatus jpIoStreamStatus, String command) Write data to session	196

Method Detail

destroyHandler

```
public void destroyHandler()
```

Destroy the session handler

---

initHandler

```
public void initHandler(JpServerSession jpServerSession)
```

Initialize the solver handler
Parameters:
`jpServerSession` - reference to server session

Appendix: JUSTSolver Java API

readData

```
public void readData(InputStream inputStream,
                     OutputStream outputStream,
                     JpIoStreamStatus jpIoStreamStatus,
                     String command)
```

Read data from session

Parameters:

`inputStream` - the input stream for reading from the client
`outputStream` - the output stream to the client
`jpIoStreamStatus` - the status of the io stream
`command` - free definable command string

---

startSession

```
public void startSession()
```

Called from jpSession.startSession();

---

writeData

```
public void writeData(InputStream inputStream,
                      OutputStream outputStream,
                      JpIoStreamStatus jpIoStreamStatus,
                      String command)
```

Write data to session

Parameters:

`inputStream` - the input stream for reading from the client
`outputStream` - the output stream to the client
`jpIoStreamStatus` - the status of the io stream
`command` - free definable command string

Package hpcc.math

Class Summary		Page
JpVector	This is a simple vector class.	*196*

Class JpVector

hpcc.math

```
java.lang.Object
  └ hpcc.math.JpVector
```

All Implemented Interfaces:
Cloneable, Comparable, Serializable

---

public class **JpVector**
extends Object
implements Serializable, Cloneable, Comparable

This is a simple vector class. The vector contains the three double components x,y,z
ATTENTION The access modifiers of the components are *public* to have a faster access on it but this is also a dangerous behaviour of this class!

Author:
Thorsten Ludewig

Field Summary		Page
double	**x** the x component of this vector	198
double	**y** the y component of this vector	199
double	**z** the z component of this vector	199

Constructor Summary	Page
JpVector() this ,,default" constructor creates a zero vector	199
JpVector(double x, double y, double z) this constructor creates the vector from the tree individual components	199
JpVector(double[] x) this constructor creates a vector form the given double array	199
JpVector(JpVector parent) create a vector from the given vector	199

Method Summary		Page
final JpVector	**add**(JpVector vector) the add method adds every component of the given vector on the corresponding component of this vector	199
int	**compareTo**(Object o) Compare this vector with an other one, Only if the vectors have identical components this method returns a 0.	200
final JpVector	**cross**(JpVector vector1) this method computes the cross product of the given vectors and stores the result in this vector	200

final JpVector	**cross**(JpVector vector1, JpVector vector2) this method computes the cross product of the given vectors and stores the result in this vector	200
double	**distance**(JpVector vector) Returns the distance to the given vector	200
final JpVector	**div**(double divisor) this method divides each component by the given divisor	201
final double	**dot**(JpVector vector) this method computes the scalar dot product of the given vector to this vector	201
JpVector	**getZeroVector3d**(double x, double y, double z) A method finding the zero point (mid point) between to vectors	201
JpVector	**getZeroVector3d**(JpVector tuple3d) A method finding the zero point (mid point) between to vectors	201
final double	**length**() this method computes the length of this vector	202
void	**max**(double x, double y, double z) Compare each single component of a second vector and stores for each component the maximum value.	202
void	**max**(JpVector tuple3d) Compare each single component of a second vector and stores for each component the maximum value.	202
void	**min**(double x, double y, double z) Compare each single component of a second vector and stores for each component the minimum value.	203
void	**min**(JpVector tuple3d) Compare each single component of a second vector and stores for each component the minimum value.	202
final JpVector	**mul**(double factor) this method multiply the given factor to each component of this vector	203
final JpVector	**sub**(JpVector vector) the sub method subtracts every component of the given vector from the corresponding component of this vector	203
String	**toString**() this method returns the String representation of this vector	203

Field Detail

x

```
public double x
```

the x component of this vector

y

```
public double y
```

the y component of this vector

---

z

```
public double z
```

the z component of this vector

Constructor Detail

JpVector

```
public JpVector()
```

this ,,default" constructor creates a zero vector

JpVector

```
public JpVector(double[] x)
```

this constructor creates a vector form the given double array

---

JpVector

```
public JpVector(JpVector parent)
```

create a vector from the given vector

JpVector

```
public JpVector(double x,
                double y,
                double z)
```

this constructor creates the vector from the tree individual components

Method Detail

add

```
public final JpVector add(JpVector vector)
```

the add method adds every component of the given vector on the corresponding component of this vector

Parameters:

`vector` - the vector to add on this vector

Returns:

this ,,result" vector

---

compareTo

```
public int compareTo(Object o)
```

Compare this vector with an other one, Only if the vectors have identical components this method returns a 0. In all others cases it returns a -1.

Parameters:

`o` - the other vector

Returns:

0 or -1

---

cross

```
public final JpVector cross(JpVector vector1)
```

this method computes the cross product of the given vectors and stores the result in this vector

Parameters:

`vector1` - the first vector

Returns:

this ,,result" vector

---

cross

```
public final JpVector cross(JpVector vector1,
                            JpVector vector2)
```

this method computes the cross product of the given vectors and stores the result in this vector

Parameters:

`vector1` - the first vector

`vector2` - the second vector

Returns:

this ,,result" vector

---

distance

```
public double distance(JpVector vector)
```

Returns the distance to the given vector
Parameters:
`vector` - the given vector
Returns:
the distance

---

div

```
public final JpVector div(double divisor)
```

this method divides each component by the given divisor
Parameters:
`divisor` - the divisor
Returns:
this ,,result" vector

---

dot

```
public final double dot(JpVector vector)
```

this method computes the scalar dot product of the given vector to this vector
Parameters:
`vector` - the vector to compute with
Returns:
the scalar result value

---

getZeroVector3d

```
public JpVector getZeroVector3d(JpVector tuple3d)
```

A method finding the zero point (mid point) between to vectors
Parameters:
`tuple3d` - the second vector
Returns:
the vector to the zero point

---

getZeroVector3d

```
public JpVector getZeroVector3d(double x,
                                double y,
                                double z)
```

A method finding the zero point (mid point) between to vectors
Parameters:
`x` - x component
`y` - y component

`z` - z component

Returns:

the vector to the zero point

---

length

```
public final double length()
```

this method computes the length of this vector

Returns:

this ,,result" vector

---

max

```
public void max(JpVector tuple3d)
```

Compare each single component of a second vector and stores for each component the maximum value.

Parameters:

`tuple3d` - the second vector

---

max

```
public void max(double x,
                double y,
                double z)
```

Compare each single component of a second vector and stores for each component the maximum value.

Parameters:

`x` - x component

`y` - y component

`z` - z component

---

min

```
public void min(JpVector tuple3d)
```

Compare each single component of a second vector and stores for each component the minimum value.

Parameters:

`tuple3d` - the second vector

---

min

```
public void min(double x,
                double y,
                double z)
```

Compare each single component of a second vector and stores for each component the minimum value.

Parameters:

`x` - x component
`y` - y component
`z` - z component

---

mul

```
public final JpVector mul(double factor)
```

this method multiply the given factor to each component of this vector

Parameters:

`factor` - the factor to multiply with

Returns:

this ,,result" vector

---

sub

```
public final JpVector sub(JpVector vector)
```

the sub method subtracts every component of the given vector from the corresponding component of this vector

Parameters:

`vector` - the subtracting vector

Returns:

this ,,result" vector

---

toString

```
public String toString()
```

this method returns the String representation of this vector

Overrides:

`toString` in class `Object`

Returns:

the three components separated by a space

Package laplacesolver3d

Class FlowVars

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.FlowVars
```

All Implemented Interfaces:

Serializable

---

public class **FlowVars**
extends Object
implements Serializable

This class contains all fields/variables that are stored in one cell and where transported to the neighbour cells. In the Laplace 3D sample it contains one field only named "mach". This type of data structure in general is called the U-vector.

Author:

Thorsten Ludewig

Field Summary		Page
double	**mach** A simple flow var field.	205

Constructor Summary	Page
FlowVars() Default constructor for FlowVars.	205
FlowVars(FlowVars vars) This constructor creates a copy of the given FlowVars object.	205

Method Summary		Page
void	**add**(FlowVars u) Add the values of the given flow vars to the current vars.	205
Object	**clone**() Clone the current FlowVars object.	206
void	**div**(double d) Divide all fields containing by the FlowVars object by the given divisor.	206
void	**mul**(double d) Multiply all fields containing by the FlowVars object by the given multiplier.	206
void	**set**(FlowVars u) Set all fields of the current FlowVars object to the same values of the given FlowVars object.	206
void	**setZero**() Set all fields zero	206
void	**sub**(FlowVars u) Subtract from the current FlowVars object fields the corresponding fields of the given FlowVars object.	207

Field Detail

mach

public double **mach**

A simple flow var field. It it named "mach" but it is a simple double number with no relation to a real Mach number.

Constructor Detail

FlowVars

public **FlowVars**()

Default constructor for FlowVars.

FlowVars

public **FlowVars**(FlowVars vars)

This constructor creates a copy of the given FlowVars object.

Method Detail

add

public void **add**(FlowVars u)

Add the values of the given flow vars to the current vars.
Parameters:
`u` - The flow vars to add on

---

clone

```
public Object clone()
```

Clone the current FlowVars object. (make a copy of it)
Overrides:
`clone` in class `Object`
Returns:
An instance of the cloned FlowVars object

---

div

```
public void div(double d)
```

Divide all fields containing by the FlowVars object by the given divisor.
Parameters:
`d` - The divisor

---

mul

```
public void mul(double d)
```

Multiply all fields containing by the FlowVars object by the given multiplier.
Parameters:
`d` - The multiplier

---

set

```
public void set(FlowVars u)
```

Set all fields of the current FlowVars object to the same values of the given FlowVars object.
Parameters:
`u` - The FlowVars object to set from

---

setZero

```
public void setZero()
```

Set all fields zero

sub

```
public void sub(FlowVars u)
```

Subtract from the current FlowVars object fields the corresponding fields of the given FlowVars object.

Parameters:

`u` - The FlowVars object

Class GlobalVars

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.GlobalVars
```

public class **GlobalVars**
extends Object

This global class is to compute the norm/residual. It it implemented as a singleton pattern.

Author:

Thorsten Ludewig

Method Summary		Page
static double	**getNorm**() Return the current norm/residual	207
static void	**setNumberOfBlocks**(int number) To initialize the GlobalVars object correct it needs to know the total number of block for the complete computation.	208
static void	**writeNorm**(double norm, int iteration) Write the norm from a block to the global norm.	208

Method Detail

getNorm

```
public static double getNorm()
```

Return the current norm/residual

Returns:

the current norm/residual

setNumberOfBlocks

```
public static void setNumberOfBlocks(int number)
```

To initialize the GlobalVars object correct it needs to know the total number of block for the complete computation.

Parameters:

`number` - number of blocks

---

writeNorm

```
public static void writeNorm(double norm,
                             int iteration)
```

Write the norm from a block to the global norm.

Parameters:

`norm` - norm from the block

`iteration` - iteration the block has finished

Class LaplaceSolver3D

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.LaplaceSolver3D
```

All Implemented Interfaces:

JpMultiblockSolver, JpSolver, Serializable

---

public class **LaplaceSolver3D**
extends Object
implements JpMultiblockSolver, Serializable

A sample implementation of a 3D Laplace solver.
The execution sequence is:

1. `Construtor: creates an instance of the solver`
2. `initSolver: initialize the solver with references to the framework`
3. `setXX: setting all solver parameters from the startup.properties file`
4. `postInitialization: computing/setting free stream conditions`
5. `solve: loop until simulation is finished`
6. `finalizeSolver: after simulation is finished`
7. `getOutputVars: for creating an output file`

Author:

Thorsten Ludewig

Field Summary	Page

FlowVars	**freeStreamConditions** The free stream condition information	*210*
double	**machNumber** The Mach number	*210*

Constructor Summary	Page
LaplaceSolver3D() Creates a new instance of LaplaceSolver3D	*210*

Method Summary		Page
void	**finalizeSolver**() This method will be executed after all computation is done.	*210*
Object	**getDataObject**(int i) For the interactive steering a client application could use this method to order specific data from the solver.	*210*
JpBoundaryCondition[]	**getFaces**() Returns an array with all face boundary conditions for this block.	*211*
Object	**getOutputVars**(int i, int j, int k) getOutputVars must return an object with public fields containing all vars for the solution output e.g TecPlot output.	*211*
void	**initSolver**(JpDomain block, int nodeId) Initialize the solver with the references to the framework	*211*
void	**postInitialization**() Is responsible for computing/setting free stream conditions	*211*
void	**setDataObject**(int i, Object object) For the interactive steering a client application could use this method to set specific data at the solver.	*212*
void	**setMachNumber**(double machNumber) Sets the Mach number (solver parameter).	*212*
void	**setMaxIterations**(int maxIteration) Sets the maximum number of iteration (solver parameter).	*212*
boolean	**solve**(int currentIteration) The ,,main" method of the solver.	*212*

Methods inherited from interface hpcc.just.share.JpMultiblockSolver
finalizeSolver, initSolver

Methods inherited from interface hpcc.just.share.JpSolver
getDataObject, getFaces, getOutputVars, setDataObject, solve

Field Detail

freeStreamConditions

```
public FlowVars freeStreamConditions
```

The free stream condition information

---

machNumber

```
public double machNumber
```

The Mach number

Constructor Detail

LaplaceSolver3D

```
public LaplaceSolver3D()
```

Creates a new instance of LaplaceSolver3D

Method Detail

finalizeSolver

```
public void finalizeSolver()
```

This method will be executed after all computation is done.

Specified by:

finalizeSolver in interface JpMultiblockSolver

---

getDataObject

```
public Object getDataObject(int i)
```

For the interactive steering a client application could use this method to order specific data from the solver. In this case it always returns *null.*

Specified by:

getDataObject in interface JpSolver

Parameters:

i - A tag specified by the developer of a solver

Returns:

the ordered data object

---

getFaces

```
public JpBoundaryCondition[] getFaces()
```

Returns an array with all face boundary conditions for this block. In this sample it is not needed.

Specified by:

getFaces in interface JpSolver

Returns:

null

---

getOutputVars

```
public Object getOutputVars(int i,
                            int j,
                            int k)
```

getOutputVars must return an object with public fields containing all vars for the solution output e.g TecPlot output.

Specified by:

getOutputVars in interface JpSolver

Parameters:

i - is index for GRID I
j - is index for GRID J
k - is index for GRID K

Returns:

the flow field vars object

---

initSolver

```
public void initSolver(JpDomain block,
                       int nodeId)
```

Initialize the solver with the references to the framework

Specified by:

initSolver in interface JpMultiblockSolver

Parameters:

block - reference to the block
nodeId - the unique node id

---

postInitialization

```
public void postInitialization()
```

Is responsible for computing/setting free stream conditions

---

setDataObject

```
public void setDataObject(int i,
                          Object object)
```

For the interactive steering a client application could use this method to set specific data at the solver.

Specified by:

setDataObject in interface JpSolver

Parameters:

i - A tag specified by the developer of a solver

object - data object

---

setMachNumber

```
public void setMachNumber(double machNumber)
```

Sets the Mach number (solver parameter). It is called by the framework during the processing of the startup.properties file.

Parameters:

machNumber - the Mach number

---

setMaxIterations

```
public void setMaxIterations(int maxIteration)
```

Sets the maximum number of iteration (solver parameter). It is called by the framework during the processing of the startup.properties file.

Parameters:

maxIteration - maximum number of iterations

---

solve

```
public boolean solve(int currentIteration)
            throws JpSolverException
```

The ,,main" method of the solver. It will be executed until the return value is false.

Specified by:

solve in interface JpSolver

Parameters:

currentIteration - the current iteration

Returns:

break condition

Throws:

JpSolverException - Throws an unexpected solver exception

Class Main

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.Main
```

---

public class **Main**
extends Object

This Main class is only a wrapper for hpcc.just.app.cli.Main and a shortcut for running from an IDE (Integrated Development Environment). It is normally not necessary.

Author:
Thorsten Ludewig
Version:
1.0
See Also:
`hpcc.just.app.cli.Main`

Constructor Summary	Page
`Main()`	213

Method Summary		Page
static void	`main(String[] args)` The main method to start with the JVM	213

Constructor Detail

Main

```
public Main()
```

Method Detail

main

```
public static void main(String[] args)
```

The main method to start with the JVM
Parameters:
`args` - the command line arguments

Class SimpleBoundaryCondition

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.SimpleBoundaryCondition
```

---

abstract public class **SimpleBoundaryCondition**
extends Object

This class is implementing the different boundary conditions.

Author:
Thorsten Ludewig

Constructor Summary	Page
SimpleBoundaryCondition()	214

Method Summary		Page
abstract void	**compute**(SimpleCell2 cell, SimpleCell2 neighbor, LaplaceSolver3D solver, JpVector normal) This method computes the specific boundary contition.	214
static SimpleBoundaryCondition	**getBoundaryCondition**(String conditionName) This method is called by the boundary handler the result is an object for the given type of the boundary condition.	215

Constructor Detail

SimpleBoundaryCondition

```
public SimpleBoundaryCondition()
```

Method Detail

compute

```
public abstract void compute(SimpleCell2 cell,
                             SimpleCell2 neighbor,
                             LaplaceSolver3D solver,
                             JpVector normal)
```

This method computes the specific boundary condition. Because it is abstract it must be filled out by a child class.

Parameters:

`cell` - The current cell

`neighbor` - the neighbour cell

`solver` - the solver

`normal` - the cell face normal vector

---

getBoundaryCondition

`public static SimpleBoundaryCondition` **`getBoundaryCondition`**`(String conditionName)`

This method is called by the boundary handler the result is an object for the given type of the boundary condition.

Parameters:

`conditionName` - A String representing a boundary condition type/name

Returns:

An object computing the specified boundary condition.

Class SimpleBoundaryHandler

laplacesolver3d

```
java.lang.Object
  └ laplacesolver3d.SimpleBoundaryHandler
```

All Implemented Interfaces:

JpBoundaryHandler, Serializable

---

public class **SimpleBoundaryHandler**
extends Object
implements JpBoundaryHandler

The SimpleBoundaryHandler class is responsible for setting the boundary conditions.

Author:

Thorsten Ludewig

Constructor Summary	Page
SimpleBoundaryHandler() Creates a new instance of SimpleBoundaryHandler	216

	Method Summary	Page
void	**init**(JpBlock jpBlock) Initialization of this class will be executed by the JUSTGrid framework.	216
void	**setFaces**(JpBoundaryCondition[] jpBoundaryCondition) Sets the face boundary condition information for this block	216
void	**setSimpleSolver3D**(LaplaceSolver3D solver) Sets the reference to the Laplace solver.	216

void	**updateBoundaryConditions**(int type) Before every single compute iteration this method will be executed by the JUSTGrid framework.	217

Methods inherited from interface hpcc.just.domain.JpBoundaryHandler
init, setFaces, updateBoundaryConditions

Constructor Detail

SimpleBoundaryHandler

```
public SimpleBoundaryHandler()
```

Creates a new instance of SimpleBoundaryHandler

Method Detail

init

```
public void init(JpBlock jpBlock)
```

Initialization of this class will be executed by the JUSTGrid framework.

Specified by:

init in interface JpBoundaryHandler

Parameters:

jpBlock - the parent block reference

---

setFaces

```
public void setFaces(JpBoundaryCondition[] jpBoundaryCondition)
```

Sets the face boundary condition information for this block

Specified by:

setFaces in interface JpBoundaryHandler

Parameters:

jpBoundaryCondition - the boundary conditions

---

setSimpleSolver3D

```
public void setSimpleSolver3D(LaplaceSolver3D solver)
```

Sets the reference to the Laplace solver.

Parameters:

solver - the parent solver

---

updateBoundaryConditions

```
public void updateBoundaryConditions(int type)
```

Before every single compute iteration this method will be executed by the JUSTGrid framework. For this sample the type is not nessesary.

Specified by:

updateBoundaryConditions in interface JpBoundaryHandler

Parameters:

type - The type of the boundary update.

Class SimpleCell2

laplacesolver3d

```
java.lang.Object
  └ hpcc.just.domain.JpCell
      └ laplacesolver3d.SimpleCell2
```

All Implemented Interfaces:

Cloneable, Serializable

---

public class **SimpleCell2**
extends JpCell

SimpleCell represents a single cell in the solution domain.

Author:

Thorsten Ludewig

Field Summary		Page
FlowVars	**u** All flow vars the so called U-vector	218

Fields inherited from class hpcc.just.domain.JpCell
finiteVolume, NUMBER_OF_FACES

Constructor Summary	Page
SimpleCell2 ( ) Creates a new instance of SimpleCell	218

Method Summary		Page
Object	**getData** ( ) This method must return an object containing all boundary exchange information for this cell.	218

void	**setData**(Object data) This method sets all information from a boundary exchange.	218

Methods inherited from class hpcc.just.domain.JpCell
getData, getFaceVector, getFaceVectors, getFiniteVolume, setData, setFaceVector, setFiniteVolume

Field Detail

u

```
public FlowVars u
```

All flow vars the so called U-vector

Constructor Detail

SimpleCell2

```
public SimpleCell2()
```

Creates a new instance of SimpleCell

Method Detail

getData

```
public Object getData()
```

This method must return an object containing all boundary exchange information for this cell.

Overrides:

getData in class JpCell

Returns:

the exchange information

---

setData

```
public void setData(Object data)
```

This method sets all information from a boundary exchange.

Overrides:

setData in class JpCell

Parameters:

data - the neighboring cell information

D JUSTSolver Template - Laplace 3D - Source Code

D.1 FlowVars.java

```
/*
 * FlowVars.java
 *
 * Created on October 8, 2006, 3:04 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package laplacesolver3d;

//~--- JDK imports ------------------------------------------------------------

import java.io.Serializable;

//~--- classes ----------------------------------------------------------------

/**
 * This class contains all fields/variables that are stored in one cell and
 * where transported to the neighbour cells. In the Laplace 3D sample it
 * contains one field only named "mach". This type of data structure in general
 *  is called the U-vector.
 * @author Thorsten Ludewig
 */
public class FlowVars implements Serializable
{
  /**
   * Default constructor for FlowVars.
   */
  public FlowVars()
  {
  }

  /**
   * This constructor creates a copy of the given FlowVars object.
   * @param vars The FlowVars object to copy
   */
  public FlowVars(FlowVars vars)
  {
    this.mach = vars.mach;
  }

  //~--- methods --------------------------------------------------------------

  /**
   * Add the values of the given flow vars to the current vars.
   * @param u The flow vars to add on
   */
  public void add(FlowVars u)
  {
    mach += ((FlowVars) u).mach;
  }
```

```
/**
 * Clone the current FlowVars object. (make a copy of it)
 * @return An instance of the cloned FlowVars object
 */
public Object clone()
{
  return new FlowVars(this);
}

/**
 * Divide all fields containing by the FlowVars object by the given divisor.
 * @param d The divisor
 */
public void div(double d)
{
  mach /= d;
}

/**
 * Multiply all fields containing by the FlowVars object by the given
 * multiplicator.
 * @param d The multiplicator
 */
public void mul(double d)
{
  mach *= d;
}

//~--- set methods ----------------------------------------------------------

/**
 * Set all fields of the current FlowVars object to the same values of the
 * given FlowVars object.
 * @param u The FlowVars object to set from
 */
public void set(FlowVars u)
{
  mach = u.mach;
}

/**
 * Set all fields zero
 */
public void setZero()
{
  mach = 0.0;
}

//~--- methods --------------------------------------------------------------

/**
 * Substract from the current FlowVars object fields the corresponding fields
 * of the given FlowVars object.
 * @param u The FlowVars object
 */
public void sub(FlowVars u)
{
  mach -= u.mach;
}
```

```
  //~--- fields ---------------------------------------------------------------

  /**
   * A simple flow var field. It it named "mach" but it is a
   * simple double number with no relation to a real Mach number.
   */
  public double mach;
}
```

D.2 GlobalVars.java

```
/*
 * GlobalVars.java
 *
 * Created on November 23, 2006, 12:26 PM
 *
 */
package laplacesolver3d;

/**
 * This global class is to compute the norm/residual. It it implemented as a
 * singleton pattern.
 * @author Thorsten Ludewig
 */
public class GlobalVars
{
  /** Field description */
  private final static GlobalVars singleton = new GlobalVars();

  //~--- constructors ---------------------------------------------------------

  /** Creates a new instance of GlobalVars */
  private GlobalVars()
  {
    norm = Double.MAX_VALUE;
  }

  //~--- get methods ----------------------------------------------------------

  /**
   * Return the current norm/residual
   * @return the current norm/residual
   */
  public static double getNorm()
  {
    return singleton.norm;
  }

  //~--- set methods ----------------------------------------------------------

  /**
   * To initialize the GlobalVars object correct it needs to know the total
   * number of block for the complete computation.
   * @param number number of blocks
   */
```

```
  public static void setNumberOfBlocks(int number)
  {
    singleton._setNumberOfBlocks(number);
  }

  //~--- methods --------------------------------------------------------------

  /**
   * Write the norm from a block to the global norm.
   * @param norm norm from the block
   * @param iteration iteration the block has finished
   */
  public static void writeNorm(double norm, int iteration)
  {
    singleton._writeNorm(norm, iteration);
  }

  /**
   * Method description
   *
   *
   * @param number
   */
  private synchronized void _setNumberOfBlocks(int number)
  {
    numberOfBlocks = number;
    counter = number;
    norm = 0.0;
    System.out.println("+++ GlobalVars.setNumberOfBlocks(" + number + ")");
  }

  /**
   * Method description
   *
   *
   * @param norm
   * @param iteration
   */
  private synchronized void _writeNorm(double norm, int iteration)
  {
    if (counter != -1)
    {
      this.norm += norm;
      counter--;

      if (counter == 0)
      {
        norm /= numberOfBlocks;
        counter = numberOfBlocks;
        System.out.println("iteration = " + iteration + "  norm = " + norm);
      }
    }
  }

  //~--- fields ---------------------------------------------------------------

  /** Field description */
  private int counter = -1;

  /** Field description */
```

```
    private double norm;

    /** Field description */
    private int numberOfBlocks;
  }
}
```

D.3 LaplaceSolver3D.java

```
/*
 * LaplaceSolver3D.java
 *
 * Created on September 27, 2006, 10:14 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package laplacesolver3d;

//~--- non-JDK imports --------------------------------------------------------

import hpcc.just.domain.JpBoundaryCondition;
import hpcc.just.domain.JpCell;
import hpcc.just.domain.JpDomain;
import hpcc.just.domain.structured.JpBlock;
import hpcc.just.share.JpMultiblockSolver;
import hpcc.just.share.JpSolverException;

//~--- JDK imports ------------------------------------------------------------

import java.io.Serializable;

//~--- classes ----------------------------------------------------------------

/**
 * A sample implementation of a 3D Laplace solver.
 * <br>
 *  The execution sequence is:
 *  <pre>
 *    1. Constructor: creates an instance of the solver
 *    2. initSolver: initialize the solver with references to the framework
 *    3. setXX: setting all solver parameters from the startup.properties file
 *    4. postInitialization: computing/setting free stream conditions
 *    5. solve: loop until simulation is finished
 *    6. finalizeSolver: after simulation is finished
 *    7. getOutputVars: for creating an output file
 *  </pre>
 * @author Thorsten Ludewig
 */
public class LaplaceSolver3D implements JpMultiblockSolver, Serializable
{
  /**
   * Creates a new instance of LaplaceSolver3D
   */
  public LaplaceSolver3D()
  {
```

```
  }

  //~--- methods --------------------------------------------------------------

  /**
   * This method will be executed after all computation is done.
   */
  public void finalizeSolver()
  {
    System.out.println("*** finalizeSolver() block #"
                       + this.block.getBlockNumber());
  }

  //~--- get methods ----------------------------------------------------------

  /**
   * For the interactive steering a client application could use this method to
   * order specific data from the solver. In this case it always
   * returns <i>null<i>.
   * @param i A tag specified by the developer of a solver
   * @return the ordered data object
   */
  public Object getDataObject(int i)
  {
    return null;
  }

  /**
   * Returns an array with all face boundary conditions for this block.
   * In this sample it is not needed.
   * @return null
   */
  public JpBoundaryCondition[] getFaces()
  {
    return null;
  }

  /**
   * getOutputVars must return an object with public fields
   * containing all vars for the solution output e.g TecPlot
   * output.
   *
   * @param i is index for GRID I
   * @param j is index for GRID J
   * @param k is index for GRID K
   *
   * @return the flow field vars object
   */
  public Object getOutputVars(int i, int j, int k)
  {
    FlowVars vars = (FlowVars) cells[i][j][k].getData();

    vars.add((FlowVars) cells[i][j + 1][k].getData());
    vars.add((FlowVars) cells[i][j][k + 1].getData());
    vars.add((FlowVars) cells[i][j + 1][k + 1].getData());
    vars.add((FlowVars) cells[i + 1][j][k].getData());
    vars.add((FlowVars) cells[i + 1][j + 1][k].getData());
    vars.add((FlowVars) cells[i + 1][j][k + 1].getData());
    vars.add((FlowVars) cells[i + 1][j + 1][k + 1].getData());
    vars.mul(0.125);
```

```
    return vars;
  }

  //~--- methods --------------------------------------------------------------

  /**
   * Initialize the solver with the references to the framework
   * @param block reference to the block
   * @param nodeId the unique node id
   */
  public void initSolver(JpDomain block, int nodeId)
  {
    this.block = (JpBlock) block;
    this.cells = this.block.getCells();
    this.numberOfHaloCells = this.block.getNumberOfHaloCells();
    this.numberOfCells = (this.block.getGridI() - 1)
                       * (this.block.getGridJ() - 1)
                       * (this.block.getGridK() - 1);
    I = cells.length;
    J = cells[0].length;
    K = cells[0][0].length;
    Ie = cells.length - numberOfHaloCells;
    Je = cells[0].length - numberOfHaloCells;
    Ke = cells[0][0].length - numberOfHaloCells;

    if (this.block.getBlockNumber() == 1)
    {
      GlobalVars.setNumberOfBlocks(this.block.getBlockArray().length);
    }
  }

  /**
   * Is responsible for computing/setting free stream conditions
   */
  public void postInitialization()
  {
    try
    {
      ((SimpleBoundaryHandler) this.block.getBoundaryHandler())
        .setSimpleSolver3D(this);
      freeStreamConditions = new FlowVars();
      freeStreamConditions.mach = this.machNumber;

      //
      // set free stream values over the hole solution domain
      //
      for (int i = 0; i < I; i++)
      {
        for (int j = 0; j < J; j++)
        {
          for (int k = 0; k < K; k++)
          {
            ((SimpleCell2) cells[i][j][k]).u.set(freeStreamConditions);
          }
        }
      }

      freeStreamConditions.mach = this.machNumber;
    }
```

```
    catch (Exception e)
    {
      e.printStackTrace();
      System.exit(0);
    }
  }

  //~--- set methods ----------------------------------------------------------

  /**
   * For the interactive steering a client application could use this method to
   * set specific data at the solver.
   * @param i A tag specified by the developer of a solver
   * @param object data object
   */
  public void setDataObject(int i, Object object)
  {
  }

  /**
   * Sets the Mach number (solver parameter). It is called by the framework
   * during the processing of the startup.properties file.
   * @param machNumber the Mach number
   */
  public void setMachNumber(double machNumber)
  {
    this.machNumber = machNumber;
  }

  /**
   * Sets the maximum number of iteration (solver parameter).
   * It is called by the framework during the processing of the
   * startup.properties file.
   * @param maxIteration maximum number of iterations
   */
  public void setMaxIterations(int maxIteration)
  {
    this.maxIteration = maxIteration;
  }

  //~--- methods --------------------------------------------------------------

  /**
   * The ,,main'' method of the solver. It will be executed until the return
   * value is <code>false</code>.
   * @param currentIteration the current iteration
   * @return break condition
   * @throws JpSolverException Throws an unexpected solver exception
   */
  public boolean solve(int currentIteration) throws JpSolverException
  {
    double avgNorm = 0.0;
    double norm;

    for (int i = numberOfHaloCells; i < Ie; i++)
    {
      for (int j = numberOfHaloCells; j < Je; j++)
      {
        for (int k = numberOfHaloCells; k < Ke; k++)
        {
```

```
          norm = ((SimpleCell2) cells[i][j][k]).u.mach;
          ((SimpleCell2) cells[i][j][k]).u.mach =
            (((SimpleCell2) cells[i - 1][j][k]).u.mach
             + ((SimpleCell2) cells[i + 1][j][k]).u.mach
             + ((SimpleCell2) cells[i][j - 1][k]).u.mach
             + ((SimpleCell2) cells[i][j + 1][k]).u.mach
             + ((SimpleCell2) cells[i][j][k - 1]).u.mach
             + ((SimpleCell2) cells[i][j][k + 1]).u.mach) / 6.0;
          norm = Math.abs(norm - ((SimpleCell2) cells[i][j][k]).u.mach);
          avgNorm += norm;
        }
      }
    }

    avgNorm /= numberOfCells;
    GlobalVars.writeNorm(avgNorm, currentIteration);

    return currentIteration < maxIteration;    // break condition
  }

  //~--- fields ---------------------------------------------------------------

  /** Field description */
  private int I;

  /** Field description */
  private int Ie;

  /** Field description */
  private int J;

  /** Field description */
  private int Je;

  /** Field description */
  private int K;

  /** Field description */
  private int Ke;

  /** Field description */
  private JpBlock block;

  /** Field description */
  private JpCell[][][] cells;

  /**
   * The free stream condition information
   */
  public FlowVars freeStreamConditions;

  /**
   * The Mach number
   */
  public double machNumber;

  /** Field description */
  private int maxIteration;

  /** Field description */
  private int numberOfCells;
```

```
  /** Field description */
  private int numberOfHaloCells;
}
```

D.4 Main.java

```
package laplacesolver3d;

/**
 * This Main class is only a wrapper for hpcc.just.app.cli.Main and a shortcut
 * for running from an IDE (Integrated Development Environment).
 * It is normally not necessary.
 * @author Thorsten Ludewig
 * @version 1.0
 * @see hpcc.just.app.cli.Main
 */
public class Main
{
  /**
   * The main method to start with the JVM
   * @param args the command line arguments
   */
  public static void main(String[] args)
  {
    System.out.println("*** LaplaceSolver3D ***");
    try
    {
      // Starting the real main method
      hpcc.just.app.cli.Main.main(args);
    }
    catch (Exception e)
    {
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

D.5 SimpleBoundaryConditions.java

```
/*
 * SimpleBoundaryConditions.java
 *
 * Created on September 27, 2006, 11:30 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
```

```
package laplacesolver3d;

//~--- non-JDK imports --------------------------------------------------------

import hpcc.math.JpVector;

//~--- classes ----------------------------------------------------------------

/**
 * This class is implementing the different boundary conditions.
 * @author Thorsten Ludewig
 */
public abstract class SimpleBoundaryCondition
{
  /** Field description */
  private final static SimpleBoundaryCondition singleton =
    new SimpleBoundaryCondition()
  {
    public void compute(SimpleCell2 cell, SimpleCell2 neighbor,
                        LaplaceSolver3D solver, JpVector normal)
    {
    }
  };

  /** Field description */
  private static _Inflow bcInflow;

  /** Field description */
  private static _Outflow bcOutflow;

  /** Field description */
  private static _Wall bcWall;

  //~--- static initializers --------------------------------------------------

  static
  {
    bcInflow = new _Inflow();
    bcWall = new _Wall();
    bcOutflow = new _Outflow();
  }

  //~--- methods --------------------------------------------------------------

  /**
   * This method computes the specific boundary condition. Because it is
   * abstract it must be filled out by a child class.
   * @param cell The current cell
   * @param neighbor the neighbour cell
   * @param solver the solver
   * @param normal the cell face normal vector
   */
  public abstract void compute(SimpleCell2 cell, SimpleCell2 neighbor,
                               LaplaceSolver3D solver, JpVector normal);

  //~--- get methods ----------------------------------------------------------

  /**
   * This method is called by the boundary handler the result is an object for
   * the given type of the boundary condition.
   * @param conditionName A String representing a boundary condition type/name
```

```
    * @return An object computing the specified boundary condition.
    */
  public static SimpleBoundaryCondition getBoundaryCondition(
          String conditionName)
  {
    SimpleBoundaryCondition bc = singleton;

    if ("inflow".equals(conditionName))
    {
      bc = bcInflow;
    }
    else if ("wall".equals(conditionName))
    {
      bc = bcWall;
    }
    else if ("outflow".equals(conditionName))
    {
      bc = bcOutflow;
    }

    return bc;
  }
}


/**
 * Implementation for the inflow boundary condition. It sets the free stream
 * conditions on the givens cell.
 * @author Thorsten Ludewig
 */
class _Inflow extends SimpleBoundaryCondition
{
  /**
   * This method computes the specific boundary condition. Because it is
   * abstract it must be filled out by a child class.
   * @param cell The current cell
   * @param neighbor the neigbour cell
   * @param solver the solver
   * @param normal the cell face normal vector
   */
  public void compute(SimpleCell2 cell, SimpleCell2 neighbor,
                      LaplaceSolver3D solver, JpVector normal)
  {
    cell.u.set(solver.freeStreamConditions);
  }
}


/**
 * Implementation for the outflow boundary condition.
 * @author Thorsten Ludewig
 */
class _Outflow extends SimpleBoundaryCondition
{
  /**
   * This method computes the specific boundary condition. Because it is
   * abstract it must be filled out by a child class.
   * @param cell The current cell
   * @param neighbor the neigbour cell
```

```
     * @param solver the solver
     * @param normal the cell face normal vector
     */
    public void compute(SimpleCell2 cell, SimpleCell2 neighbor,
                        LaplaceSolver3D solver, JpVector normal)
    {
      cell.u.set(neighbor.u);
    }
}


/**
 * Implementation for the wall boundary condition. In this case (Laplace with
 * Dirichlet boundary conditions) it sets the flow var to zero at the given
 * cell.
 * @author Thorsten Ludewig
 */
class _Wall extends SimpleBoundaryCondition
{
    /**
     * This method computes the specific boundary condition. Because it is
     * abstract it must be filled out by a child class.
     * @param cell The current cell
     * @param neighbor the neigbour cell
     * @param solver the solver
     * @param normal the cell face normal vector
     */
    public void compute(SimpleCell2 cell, SimpleCell2 neighbor,
                        LaplaceSolver3D solver, JpVector normal)
    {
      cell.u.setZero();

      // cell.u.set( neighbor.u );
    }
}
```

D.6 SimpleBoundaryHandler.java

```
/*
 * SimpleBoundaryHandler.java
 *
 * Created on September 28, 2006, 9:48 AM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package laplacesolver3d;

//~--- non-JDK imports --------------------------------------------------------

import hpcc.just.domain.JpBoundaryCondition;
import hpcc.just.domain.JpCell;
import hpcc.just.domain.JpFace;
import hpcc.just.domain.structured.JpBlock;
```

```
//~--- classes ----------------------------------------------------------------

/**
 * The SimpleBoundaryHandler class is responsible for setting the boundary
 * conditions.
 * @author Thorsten Ludewig
 */
public class SimpleBoundaryHandler
        implements hpcc.just.domain.JpBoundaryHandler
{
  /**
   * Creates a new instance of SimpleBoundaryHandler
   */
  public SimpleBoundaryHandler()
  {
  }

  //~--- methods --------------------------------------------------------------

  /**
   * Initialization of this class will be executed by the JUSTGrid framework.
   * @param jpBlock the parent block reference
   */
  public void init(JpBlock jpBlock)
  {
    System.out.println("Init BoundaryHandler for block "
                       + jpBlock.getBlockNumber());
    block = jpBlock;
  }

  //~--- set methods ----------------------------------------------------------

  /**
   * Sets the face boundary condition information for this block
   * @param jpBoundaryCondition the boundary conditions
   */
  public void setFaces(JpBoundaryCondition[] jpBoundaryCondition)
  {
    System.out.println(">> setFaces BoundaryHandler ");
  }

  /**
   * Sets the reference to the Laplace solver.
   * @param solver the parent solver
   */
  public void setSimpleSolver3D(LaplaceSolver3D solver)
  {
    this.solver = solver;
    this.cells = this.block.getCells();
    I = cells.length;
    J = cells[0].length;
    K = cells[0][0].length;
    this.numberOfHaloCells = this.block.getNumberOfHaloCells();
  }

  //~--- methods --------------------------------------------------------------

  /**
   * Before every single compute iteration this method will be executed by the
```

```
 * JUSTGrid framework. For this sample the type is not necessary.
 * @param type The type of the boundary update.
 */
public void updateBoundaryConditions(int type)
{
  // System.out.println( ">> updateBoundaryConditions BoundaryHandler " );
  try
  {
    for (int face = 1; face <= this.block.NUMBER_OF_FACES; face++)
    {
      String boundaryCondition =
        this.block.getFace(face).getFacePart(1).getBoundaryCondition();
      SimpleBoundaryCondition bc =
        SimpleBoundaryCondition.getBoundaryCondition(boundaryCondition);

      switch (face)
      {
        case 1 :    // K-min
          for (int i = 0; i < I; i++)
          {
            for (int j = 0; j < J; j++)
            {
              for (int h = 0; h < numberOfHaloCells; h++)
              {
                bc.compute((SimpleCell2) cells[i][j][h],
                           (SimpleCell2) cells[i][j][h + 1], solver,
                           cells[i][j][h].getFaceVector(JpFace.K_MAX));
              }
            }
          }

        break;

      case 2 :    // J-min
        for (int i = 0; i < I; i++)
        {
          for (int k = 0; k < K; k++)
          {
            for (int h = 0; h < numberOfHaloCells; h++)
            {
              bc.compute((SimpleCell2) cells[i][h][k],
                         (SimpleCell2) cells[i][h + 1][k], solver,
                         cells[i][h][k].getFaceVector(JpFace.J_MAX));
            }
          }
        }

        break;

      case 3 :    // I-min
        for (int j = 0; j < J; j++)
        {
          for (int k = 0; k < K; k++)
          {
            for (int h = 0; h < numberOfHaloCells; h++)
            {
              bc.compute((SimpleCell2) cells[h][j][k],
                         (SimpleCell2) cells[h + 1][j][k], solver,
                         cells[h][j][k].getFaceVector(JpFace.I_MAX));
```

```
        }
      }
    }

    break;

  case 4 :     // I-max
    for (int j = 0; j < J; j++)
    {
      for (int k = 0; k < K; k++)
      {
        for (int h = 0; h < numberOfHaloCells; h++)
        {
          bc.compute(
              (SimpleCell2) cells[I - 1 - h][j][k],
              (SimpleCell2) cells[I - 2 - h][j][k], solver,
              cells[I - 1 - h][j][k].getFaceVector(JpFace.I_MIN));
        }
      }
    }

    break;

  case 5 :     // J-max
    for (int i = 0; i < I; i++)
    {
      for (int k = 0; k < K; k++)
      {
        for (int h = 0; h < numberOfHaloCells; h++)
        {
          bc.compute(
              (SimpleCell2) cells[i][J - 1 - h][k],
              (SimpleCell2) cells[i][J - 2 - h][k], solver,
              cells[i][J - 1 - h][k].getFaceVector(JpFace.J_MIN));
        }
      }
    }

    break;

  case 6 :     // K-Max
    for (int i = 0; i < I; i++)
    {
      for (int j = 0; j < J; j++)
      {
        for (int h = 0; h < numberOfHaloCells; h++)
        {
          bc.compute(
              (SimpleCell2) cells[i][j][K - 1 - h],
              (SimpleCell2) cells[i][j][K - 2 - h], solver,
              cells[i][j][K - 1 - h].getFaceVector(JpFace.K_MIN));
        }
      }
    }

    break;

  default :
    System.err.println("Unknown face");
    System.exit(0);
```

```
      }
    }
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

//~--- fields ---------------------------------------------------------------

/** Field description */
private int I;

/** Field description */
private int J;

/** Field description */
private int K;

/** Field description */
private JpBlock block;

/** Field description */
private JpCell[][][] cells;

/** Field description */
private int numberOfHaloCells;

/** Field description */
private LaplaceSolver3D solver;
}
```

D.7 SimpleCell.java

```
/*
 * SimpleCell.java
 *
 * Created on September 27, 2006, 9:51 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package laplacesolver3d;

/**
 * SimpleCell represents a single cell in the solution domain.
 * @author Thorsten Ludewig
 */
public class SimpleCell2 extends hpcc.just.domain.JpCell
{
  /**
   * Creates a new instance of SimpleCell
   */
  public SimpleCell2()
  {
```

```
    u = new FlowVars();
  }

  //~--- get methods ----------------------------------------------------------

  /**
   * This method must return an object containing all boundary exchange
   * information for this cell.
   * @return the exchange information
   */
  public Object getData()
  {
    return u.clone();
  }

  //~--- set methods ----------------------------------------------------------

  /**
   * This method sets all information from a boundary exchange.
   * @param data the neighbouring cell information
   */
  public void setData(Object data)
  {
    u = (FlowVars) data;
  }

  //~--- fields ---------------------------------------------------------------

  /**
   * All flow vars the so called U-vector
   */
    public FlowVars u;
  }
```

D.8 JUSTGrid source code statistics

	lines of code	files	packages	methods
JUSTGrid framework	21281	98	28	565
GRXMonoblock 2D	7817	45	35	362
GRXMonoblock 3D	5232	35	15	236
GRX 2D	4276	18	7	115
GRX 3D	5512	26	8	144
Showme 3D (VVT)	10735	70	8	313
ControlCenter	1674	6	1	51
CLI	795	3	1	20
Samples	10169	43	12	421
Tests	7034	33	9	247
Solver Euler 3D	1429	8	1	46
Solver Laplace 3D	1003	7	1	40
Sum	**76957**	**392**	**126**	**2560**

E JUSTCube

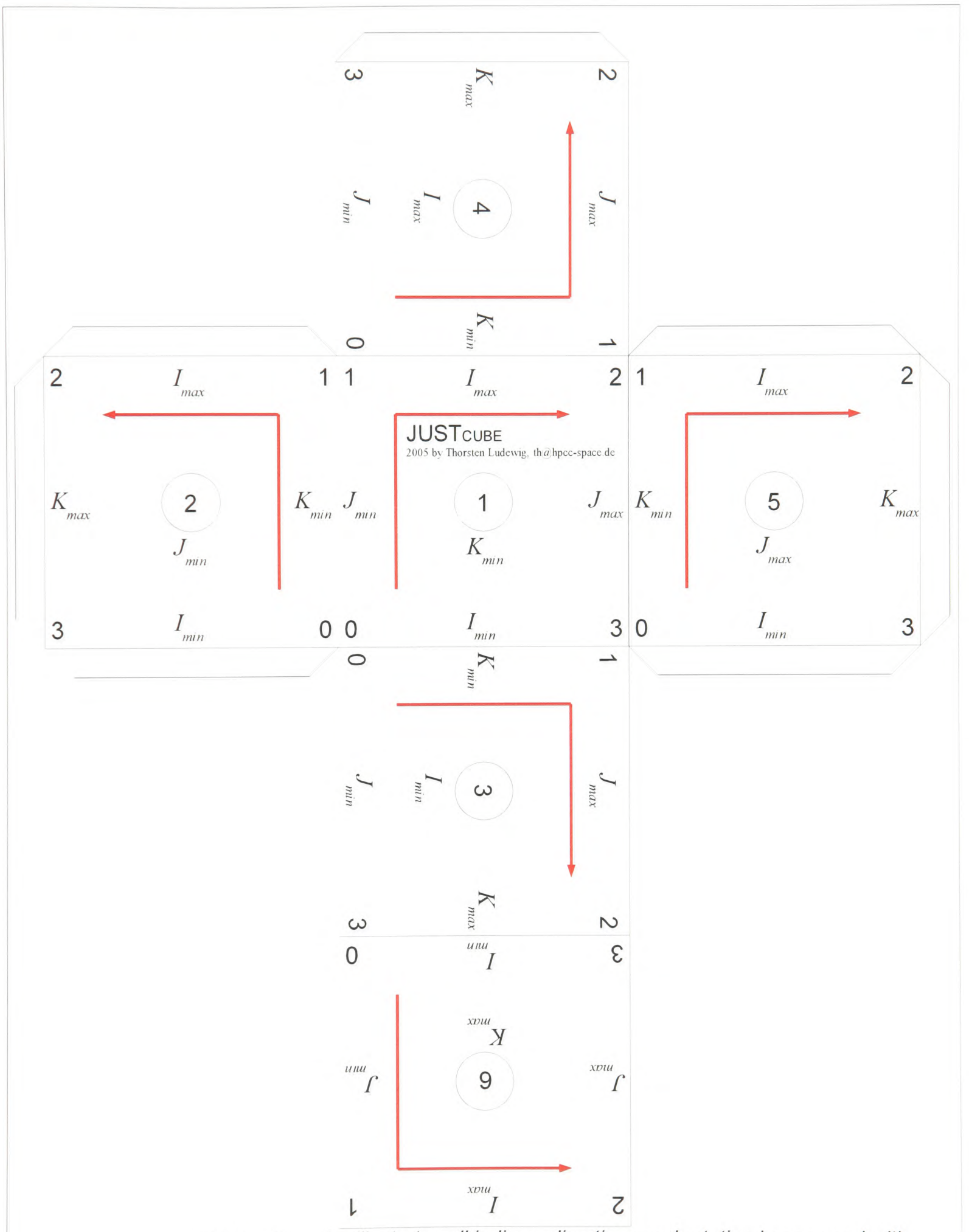


Illustration 9.1: JUSTCube this cube illustrates all indices, directions and rotation bases used with JUSTGrid

Bibliography

TRI00: Tristram, Clair, *Supercomputing Resurrected*, MIT Technology Review,2003

HPPR: Heinz-Otto Peitgen, Peter H. Richter, *The Beauty of Fractals. Images of Complex Dynamical Systems*, 1986,Springer, Berlin,978-3540158516

SHA01: Shang J.S., *Resent Research in Magneto-aerodynamics*, Progress in Aerospace,2001

SHA02: Shang J.S., *Shared Knowledge in Computational Fluid dynamics, electromagnetics, and Magneto-aerodynamics*, Progress in Aerospace,2002

BAT01: Batten, N. Clarke, C. Lambert and D. Causon, *On the choice of wave Speeds for the HLLC Rieman Solver*, Sci. Phys,1988

TRR01: M. Torrilhon, *Exact Solver and Uniqueness Conditions for Riemann Problem of Ideal Magnetohydrodynamics*, Eidgenoessische Technishe Hochshule,2002

PBU01: Pamela Walatka, Pieter Buning, Larry Pierce, Patricia Elson, *PLOT3D User's Guide*, 1990,NASA,NASA TM 101067

GRP01: Program Development Company, *Home page*, 2006, `www.gridpro.com`

HAU01: Häuser, J., Ludewig, Th., Gollnick, T., Winkelmann, R., Williams, R., D., Muylaert, J., Spel, M., A Pure Java Parallel Flow Solver,37th AIAA Aerospace Sciences Meeting and Exhibit,Reno, Nevada, USA,January, 11-14,1999,AIAA-1999-0549

TPL01: Tecplot Inc., *Home Page*, 2006, `www.tecplot.com`

HAU02: Häuser,J., Ludewig, T., Williams, Roy D., Winkelmann, R., Gollnick, T., Brunett, S., Muylaert, J., NASA Panel Java Soundbytes,5th National Symposium on Large-Scale Analysis, Design and Intelligent Synthesis Environments,Williamsburg, VA, USA,October, 12-15,1999,

HAU03: Häuser,J., Ludewig, T., Williams, Roy D., Winkelmann, R., Gollnick, T., Brunett, S., Muylaert, J., A Test Suite for High Performance Parallel Java,5th National Symposium on Large-Scale Analysis, Design and Intelligent Synthesis Environments,Williamsburg, VA, USA,October, 12-15,1999,

HAU04: Häuser,J., Ludewig, T., Williams, Roy D., Winkelmann, R., Gollnick, T., Brunett, S., Muylaert, J., *A Test Suite for High Performance Parallel Java*, Advances in Engineering Software,2000

HAU05: Häuser, J., Ludewig, T., Gollnick, T., Williams, Roy D., Javagrid: An Innovative Software for HPCC,A Paper for Computational Fluid Dynamics Conference,Swansea, UK,September,2001,

HAU06: Häuser, J., Ludewig, T., Paap, H.-G., Muylaert, J.-M., Numerical Modeling of Divergence Constraints for MHD Equations on Curvilinear Grids,Proceedings of MASCOT 07, 7th Meeting on applied scientific computing and tools,Roma, Italy,13-14 September,2007,ISSN 1098-870X

LUD01: Ludewig, T., Häuser, J., Gollnick, T., Paap, H.-G, JUST GRID: A Pure Java HPCC Grid Architecture for Multi-Physics Solvers using Complex Geometries,42nd AIAA Aerospace Science Meeting and Exhibit,Reno, Nevada, USA,January, 5-8,2004,AIAA-2004-1091

LUD02: Ludewig, T., Häuser, J., Gollnick, T., Dai, W., Paap, H., A Java Based High Performance Solver for Hierachical Parallel Computer Architectures,43rd AIAA Aerospace Science Meeting and Exhibit,Reno, Nevada, USA,January, 11-13,2005,AIAA-2005-1383

LUD03: Ludewig, T., Papadopulus, P., Häuser, J., Gollnick, T., Dai, W., Muylaert, J.-M., Paap, H., JUSTGrid A Pure Java HPCC Grid Architecture for Multi-Physics Solvers Performance and efficiency results from various Java solvers.,45th AIAA Aerospace Science Meeting and Exhibit,Reno, Nevada, USA,January, 8-11,2007,AIAA-2007-1112

TOR01: E., F., Toro, *Rienmann Solvers and Numerical Methods for Fluids Dynamics*, 1999,Springer,

FOS01: Foster, Ian, *The Grid: Computing without Bounds, Scientific American*, Scientific American,2003

FOS02: Foster, Ian, *The Grid: Blueprint for a new Computing Infrastructure*, Morgan Kaufmann,1999

GIN01: 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*, Elsevier,2000

HOR01: Horstman, Cay S., Cornell, G., *Core JAVA, Volume I-Fundamentals*, 2000,Prentice Hall,

HOR02: Horstman, Cay S., Cornell, G., *Core JAVA, Volume II-Advanced Features*, 2000,Prentice Hall,

MOR01: Moreira, J.E., S. P. Midkiff, M. Gupta, *A Comparison of Java, C/C++, and Fortran for Numerical Computing*,IBM ,2002,IBM Research Report RC 21255

MOR02: Moreira, J.E., S. P. Midkiff, M. Gupta, *From Flop to Megaflop: Java for Technical Computing*,IBM ,2002,IBM Research Report RC 21166

SCI01: Scientific Computing World, *The Need for Software*, Scientific Computing World,2000

WIN01: Winkelmann, R., Häuser J., Williams R.D, *Strategies for Parallel and Numerical Scalability of CFD Codes*, Comp. Meth. Appl. Mech. Engng., NH-Elsevier,1999

TTH01: G. Toth, *The Divergence B = 0 Constraint in Shock-Capturing Magnetohydrodynamics Codes*, Journal of Computational Physics,2000

Alphabetical Index