TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Computation in Engineering

A High-Performance Computational Steering Framework for Engineering Applications

Jovana Knežević

Vollständiger Abdruck der von der Ingenieurfakultät Bau Geo Umwelt der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr.-Ing. A. Borrmann
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. rer. nat. E. Rank
	2.	Univ.-Prof. Dr. rer. nat. habil. H.-J. Bungartz
	3.	Prof. Dr. C. R. Johnson, The University of Utah / USA

Die Dissertation wurde am 15. 04. 2013 bei der Technischen Universität München eingereicht und durch die Ingenieurfakultät Bau Geo Umwelt am 21. 07. 2013 angenommen.

Abstract

Exploration tools with sophisticated user interfaces are necessary to support experts in analytical and decision-making processes, facilitating their comfortable interaction with the complex, simulated phenomenon at hand. Such tools typically consist of a back-end simulation component, running sometimes on a high-performance cluster and a user interface front-end that lets the researcher follow intuitively interpretable visualised numerical results and immediately react based on the trends of the running computation. To achieve this, the connection between a user interaction and its effect on the simulation must not be lost. The main challenge, then, becomes to instantly and seamlessly feed the user requirements back into the computation. The state-of-the-art solutions for this may perform well for particular targeted scenarios; however, they are mostly limited in their possible application, or entail heavy code changes in order to work in conjunction with the existing code.

To leverage this concept for a broader scope of scenarios, this study proposes a generic integration framework, aimed for engineering applications, which – with minimal code changes – supports distributed computation, as well as visualisation on-the-fly. In order to reduce latency and enable a high degree of interactivity, the regular course of the simulation coupled to the framework is interrupted in small, cyclic intervals followed by a check for updates. When new data is received, the simulation automatically restarts at once with the updated settings. To obtain rapid, albeit approximate, feedback from the simulation in the case of perpetual user interaction, it is useful to apply multi-hierarchical approaches combined with simulation parallelisation. This is investigated in the context of an optimal application basis, and hence, is well-supported by the framework. In addition, the framework tackles the problem of long communication delays caused by huge data sets, which can hinder the user in exploring the effect of his interaction.

The proposed framework is applied to a number of environments, including an application for exploring temperature distribution, a neutron transport simulation, that enables researchers' interplay with virtual models of nuclear reactors under various conditions and the trial integration into the sophisticated *SCIRun* Problem Solving Environment itself to extend this environment for large-scale problems. The framework is also integrated into an existing pre-operative planning environment for joint replacement surgery, enabling a real-time interactive patient-specific selection of the optimal implant design, size, and position, for even higher accuracy in computations.

Zusammenfassung

Um Experten aus dem Bereich Computational Science and Engineering bei der visuellen Datenanalyse zu unterstützen, sind ausgefeilte interaktive Explorationswerkzeuge notwendig. Diese Werkzeuge haben i. d. R. anspruchsvolle Benutzeroberflächen, die eine möglichst einfache und komfortable Interaktion mit den zu Grunde liegenden komplexen, simulierten Phänomenen anbieten sollen. Sie bestehen typischerweise aus einer Back-End-Simulationskomponente, die oftmals auf Hochleistungsrechnern läuft, und einem visuellen Front-End als Benutzeroberfläche, mittels der ein Anwender intuitiv die Berechnungsergebnisse erfassen sowie unmittelbar auf die Tendenz der laufenden Berechnung reagieren kann. Um dieses Ziel zu erreichen, muss aber der Zusammenhang zwischen Ursache und Wirkung unbedingt erhalten bleiben, d. h. die Simulation muss in Echtzeit auf Änderungen reagieren und dem Anwender neue Berechnungsergebnisse liefern. Nur so kann das Empfinden von echtem interaktiven Arbeiten aufkommen.
Die größte Herausforderung besteht nun darin, die vom Anwender getätigten Änderungen sofort wieder in die Berechnungen einfließen zu lassen. State-Of-The-Art-Lösungen bieten heutzutage für ausgewählte Szenarien bereits gute Ergebnisse, haben jedoch entweder eine begrenzte Anwendungsmöglichkeit oder erfordern zahlreiche Änderungen des vorhandenen Codes. Damit dieses Konzept für einen breiteren Anwendungsbereich von Szenarien genutzt werden kann, ist ein generisches Integrationsframework als Teil dieser Arbeit entwickelt worden. Das Framework ist auf technische Anwendungsfälle ausgelegt und unterstützt verteiltes Rechnen sowie Online-Visualisierung mit minimalen Code-Änderungen. Der Verlauf einer Simulation, die mit diesem Framework gekoppelt wurde, wird dabei in kleinen, zyklischen Abständen unterbrochen, gefolgt von einer Prüfung auf Änderungen (z. B. neue Randbedingungen oder neue Lage/Art von geometrischen Objekten). Falls neue Daten empfangen wurden, startet die Simulation anschließend automatisch mit den aktualisierten Einstellungen. Multihierarchische Ansätze kombiniert mit effizienten Parallelisierungsstrategien sind hierbei von Vorteil, falls Nutzer ein schnelles, wenn auch nur qualitatives Feedback der Simulation benötigen. Eine (ggf. nicht interaktive) quantitative Analyse ist darüber hinaus jederzeit ebenfalls möglich. Weiterhin behandelt das Framework Probleme durch Kommunikationsverzögerungen aufgrund großer Datenmengen, die ein interaktives Arbeiten behindern.
Das vorgestellte Framework wird in dieser Arbeit auf eine Reihe von unterschiedlichen Problemstellungen angewendet, um damit das Verhältnis von Kosten und Nutzen einer interaktiven Datenexploration aufzuzeigen. Die einzelnen Problemstellungen werden jeweils im Hinblick auf die notwendigen Code-Änderungen

sowie die nunmehr vorhandenen Möglichkeiten zur Interaktion untersucht. Als Beispiele wurden dafür einfache bis komplexe Szenarien aus dem Ingenieurwesen und den Naturwissenschaften gewählt, diese umfassen u. a. eine Neutronentransportsimulation, die Industriestandardsoftware SCIRun sowie ein Werkzeug aus der bio-medizinischen Anwendung für die prä-operative Planung und Analyse von Hüftgelenksimplantationen.

Acknowledgements

This work has been done at the Chair for Computation in Engineering at Technische Universität München (TUM). I would like to gratefully acknowledge the finance from the Munich Centre of Advanced Computing (MAC) and the International Graduate School of Science and Engineering at TUM.

I must first express my gratitude to my first supervisor Prof. Dr. Ernst Rank, who not only offered me an opportunity to do scientific research in my field of interest, but also inspired and guided it, and encouraged me to participate in seminars and many international conferences.

I would like to thank Prof. Dr. Hans-Joachim Bungartz for all his motivating talks I had the privilege to hear, his agreement to review and evaluate the thesis and to be an examiner.

I would like to express appreciation to Prof. Dr. Chris R. Johnson for inviting me to the Scientific Computing and Imaging (SCI) Institute for three months to work in his group, which has been a wonderful and rewarding experience. Moreover, I am greateful to him for reviewing the thesis and accepting the request to be an examiner.

I would like to express my very special thanks to my supervisor and team leader Dr. Ralf-Peter Mundani for fruitful discussions and support during my PhD and Master studies, and both him and Dr. Martin Ruess for "bringing me" to Munich and guiding my 3-month research project back in 2008 within the SimLab Program. Our group Efficient Algorithms – especially my office-mate Jerômé Frisch – infused my working days with their enthusiasm. Collaborators from the University of Utah – Prof. Dr. Tatjana Jevremović, Dr. Hermilo Hernández, Thomas Fogal and Ayla Khan, as well as all the members of the MAC B8 Team – have also rounded out this experience. I am also grateful to Thomas Fogal and Jeremiah Hendren for their efforts in text editing.

Thanks to friends and colleagues for their support, in particular Hanne Cornils, Iason Papaioannou, Milica Grahovac and the families Witzig, Kozica and Coburn; and especially to Christian Sorger for motivating and inspiring me in the last year.

The indispensable embrace of my whole family – especially my parents Ljiljana and Dragan and my grandmother Nada – made it possible for me to focus only on my own tasks. Their unconditional encouragement means the world to me.

Jovana Knežević

Munich, April 2013

Contents

1 Introduction **1**
1.1 Numerical Simulation 1
1.2 Computational Steering 2
1.3 Classifications of Computational Steering Systems 3
1.3.1 User Interfaces 4
1.3.2 Aim of Steering 4
1.3.3 Type of Steering 5
1.3.4 Level of Interactivity 5
1.3.5 Support for Distributed Computing 6
1.3.6 Support for Multi-Component and Collaborative Steering . 6
1.3.7 Utilisation Method 6
1.4 Computational Steering in Applications 7
1.4.1 Civil Engineering and Architecture 7
1.4.2 Biomedical and Biology Applications 8
1.4.3 Physics-Based Simulations 10
1.4.4 Pollution Modelling 11
1.4.5 Nuclear Engineering 12
1.4.6 Steered Molecular Dynamics 12
1.4.7 Manufacturing 13
1.4.8 Other Application Areas 13
1.5 Real-Time Interactive Computing Environment 15
1.6 Task Definition and Approach 15
1.7 Guide Through Chapters 17

2 Interactive Computing **19**
2.1 Challenges in Building a Scalable Interactive Computing Environment 20
2.2 Efficient Simulation Basis 22
2.2.1 Approximation 22
2.2.2 Solvers 27
2.3 High Performance Computing (HPC) 31
2.3.1 State of the Art Parallel Computer Architectures 32
2.3.2 Parallel Programs 34
2.3.3 Decomposition 35
2.3.4 Mapping of the Tasks 40
2.3.5 Load Balancing (LB) 42
2.3.6 Parallelisation and Communication APIs, Libraries and Middleware 42

2.3.7 Examples . . . 45
2.4 Postprocessing and Communication . . . 47
2.4.1 Network Hardware . . . 50
2.4.2 Challenges . . . 50
2.5 Visualisation . . . 51
2.5.1 GPU Hardware . . . 51
2.5.2 GPU Computing . . . 51
2.5.3 Scientific Visualisation . . . 52
2.6 Software Engineering . . . 54
2.6.1 Design Patterns . . . 54
2.6.2 Dataflow Model – Advantages for Interactive Steering . . . 54
2.7 State-of-the-Art Interactive Steering Environments . . . 55
2.7.1 Frameworks . . . 55
2.7.2 Toolkits . . . 55
2.7.3 PSEs, Environments and Systems . . . 56
2.7.4 Libraries . . . 57
2.7.5 Collaborative Steering . . . 57
2.8 Projects within the Chair for Computation in Engineering . . . 59
2.9 Difference of our Approach . . . 60
2.10 Conclusions . . . 60

3 Implementation 63
3.1 The Concept of the Framework . . . 63
3.1.1 Signals Introduction . . . 65
3.1.2 The Concept of the Framework . . . 74
3.1.3 Challenges . . . 78
3.1.4 Example . . . 83
3.2 Framework and Multithreading . . . 84
3.2.1 Signals and Multithreading . . . 85
3.2.2 The Concept of the Framework . . . 85
3.2.3 Challenges . . . 86
3.2.4 Example . . . 88
3.3 Framework and Distributed/Hybrid Scenario . . . 88
3.3.1 Signals and Distributed scenario . . . 90
3.3.2 The Concept of the Framework . . . 90
3.3.3 Challenges . . . 90
3.3.4 Example . . . 92
3.4 Suggested Hierarchical Approaches . . . 93
3.4.1 H-refinement . . . 95
3.4.2 Up- and Down-Sampling . . . 96
3.4.3 P-refinement . . . 97
3.5 Data Transfer . . . 99
3.5.1 Our Framework Functionality for Communication . . . 99
3.5.2 Fast Data Transfer . . . 99
3.6 Visualisation and GUI Front-End . . . 100
3.6.1 Challenges . . . 100
3.7 Results . . . 102

3.7.1 Overhead of the Framework 102
3.7.2 Comparison of the Results for Linear and Hierarchical Broadcast 103
3.8 Conclusions 103

4 Application 111
4.1 Interactive Computing in Engineering Applications 111
4.2 2D Heat Conduction Simulation 112
4.2.1 Problem Description 112
4.2.2 Simulation 113
4.2.3 Interactive Visualisation 113
4.2.4 Communication Pattern 113
4.2.5 Initial Settings 114
4.2.6 User Interaction 114
4.2.7 The Framework in Action 114
4.2.8 Hierarchical Approach 115
4.2.9 Results 118
4.2.10 Effort to Integrate the Framework 118
4.3 AGENT Software 119
4.3.1 Problem Description 119
4.3.2 Simulation 119
4.3.3 Interactive Visualisation 120
4.3.4 Communication Pattern 120
4.3.5 Initial Settings 120
4.3.6 User Interaction 120
4.3.7 The Framework in Action 121
4.3.8 Hierarchical Approach 121
4.3.9 Results 122
4.3.10 Effort to Integrate the Framework 125
4.4 The SCIRun Software 126
4.4.1 Problem Description 126
4.4.2 Simulation 126
4.4.3 Interactive Visualisation 127
4.4.4 Communication Pattern 127
4.4.5 Initial Settings 128
4.4.6 User Interaction 128
4.4.7 The Framework in Action 128
4.4.8 Results 130
4.4.9 Effort to Integrate the Framework 132
4.5 Virtual Planning of Hip-Joint Surgeries – "Bone" 133
4.5.1 Problem Statement 133
4.5.2 Simulation 133
4.5.3 Interactive Visualisation 134
4.5.4 Communication 135
4.5.5 The Framework in Action 135
4.5.6 Hierarchical Approach 138
4.5.7 Parallelisation for Hierarchically Organised Tasks 141
4.5.8 Results 150

4.5.9 Effort to Integrate the Framework 155
4.6 Conclusions 157
4.6.1 Hierarchical Approaches 159
4.6.2 Effort to Integrate the Framework 159

5 Summary 161
5.1 Conclusions 162
5.2 Outlook 163

Appendix: Code Modifications 165

Bibliography 171

Chapter 1

Introduction

1.1 Numerical Simulation

Numerical simulation opens the door for scientists and engineers to provide an insight into a problem, which cannot be gained by theory or real experiments. Real experiments cannot always be done. Sometimes they are not affordable. Even if they are affordable, they might not be ethical (many examples may be found in medicine, for instance), might not be safe, or reasonable (such as explosions, or flooding of a terrain). But there is even more to it – often, it would be desirable to *predict* certain phenomena (such as weather), or natural disasters (such as tsunami). In combination with sophisticated scientific visualisation representation methods, scientists and engineers see their simulations as an opportunity to gain a better insight into various phenomena within their field of expertise – to get a better mental vision.

The practice of simulating physical phenomena was limited in the past by several factors – from the objective ones, such as memory capacity of the computing systems, to the subjective ones, such as impatience of scientists, for example. With new hardware technologies, efficient algorithms, data structures, and parallelisation strategies, this scientific simulation of very complex problems, which used to be beyond belief, has become a realistic endeavour.

Namely, most of the problems the numerical community is interested in are built around real-world phenomena. Those usually commonly known processes are driven by well-established systems of equations. Thus, scientific simulation requires the ability to model a desired physical problem domain, altogether with appropriate boundary conditions, and do numerical approximations of the governing system of equations of a particular phenomenon. The result is then validated and, in the ideal case, visualised for more intuitive interpretations. These are all rather complex steps with data interdependences, often executed as separate programs with their own data formats, hence, often require computationally expensive conversion steps in order to amalgamate.

Constructing a model means finding a mathematical model to a physical system and, thus, an approximate description of the physical world, as well as geometrical construction of a physical problem domain, in which a continuous structure is discretised, approximating the actual domain. In addition to approximating the geometry, some (idealised) physical properties, such as material properties, can be associated with the discrete domain. This typically time-consuming phase often has to be repeated for each new configuration.

For *numerical approximations* of the governing ordinary or partial differential equations based on the chosen discretisation and corresponding boundary (heat on the boundary of a heat conduction simulation, e. g.), and initial conditions (how the heat sources inside the domain are distributed, e. g.), one can use either common approximation methods (the Finite Element Method (FEM), the Finite Difference Method (FDM), Finite Volume Method (FVM), etc.), or even combine problems on multiple scales, as proposed methodologies in [172] and [122].

This yields a linear or a non-linear system of equations, which is solved using either the existing direct or iterative methods (with or without preconditioning), or hierarchical methods (multigrid and relatives), or even a combination of these if appropriate. The size of the system can vary significantly – from hundreds or thousands to millions or billions of unknowns – thus, the corresponding computation time may vary significantly.

The final part of a conventional simulation flow – efficient graphical representation of the resulting, often large, data sets – is itself a considerable task.

Aforementioned cycles are traditionally carried out as a sequence of computations (Fig. 1.1, left). Still, the wide range of experts in developing fields has to be kept in the centre of analysis and, thus, needs an interactive and, moreover, *collaborative* exploration environments.

1.2 Computational Steering

Interactive computing in general refers to user interplay with a program during its run time in order to estimate its current state and/or tendency and to react on its progress. In the numerical community in particular, an expert would like to intervene with the *running* simulation so as to change some parameter settings, geometry, boundary conditions, etc. This concept is widely known as *computational steering* (Fig. 1.1, right).

The concept is present in the scientific and engineering communities for many years now. The pioneer of a related idea was *interactive visual computing* – a process whereby scientists communicate with data by manipulating its visual representation during processing [134, 62].

The more sophisticated process of navigation allows scientists to steer or dynamically influence the simulated phenomena while they are occurring. In the Panel Report on Visualization in Scientific Computing Workshop (ViSC) [62] it is re-

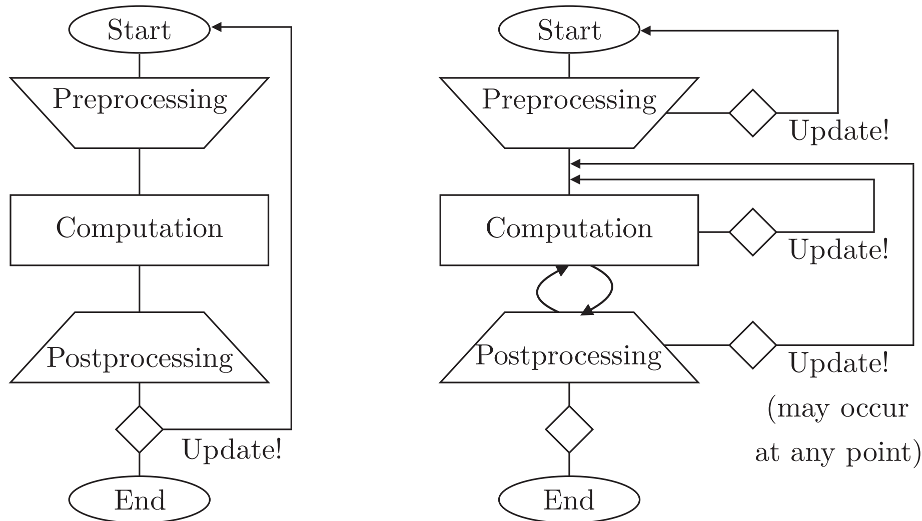


Figure 1.1: Difference of algorithmic schemas for: left – traditional simulation cycles (an update can be applied only when the previous computation is finished); right – the concept of computational steering (an update can be recognised at any point, and the computation restarts accordingly).

ported that the "scientists not only want to analyse data that results from super-computations; they also want to interpret what is happening to the data *during* super-computations. Researchers want to steer calculations in close-to-real-time; they want to be able to change parameters, resolution and representation, and see the effects; they want to drive the scientific discovery process; they want to interact with their data." Although it has been a quarter of a century since the report was written, these conclusions have remained the driving forces for the development of real-time, or close-to-real-time, steering systems.

However, the "definitions" of the *interactive computational steering tool* have been changing over time, i. e. what the term exactly refers to and when it can be used. On the one hand, the minimal user requirements or expectations are changing, and on the other, the criteria dictated by developers and the features they *choose* to support. Even nowadays, in modern applications, those criteria vary significantly. After the thorough study of the existing steering tools and environments, the author dares to state that the term "computational steering" is certainly *not* self-explanatory in any particular sense. Thus, the following section provides a further taxonomy.

1.3 Classifications of Computational Steering Systems

The classification suggested in this thesis is inspired mostly by the analysis and taxonomies in [135], [168] and [127]. According to these taxonomies, different steering tools are roughly differentiated among depending on:

- user interfaces,
- aim of steering,
- type of steering,
- level of interactivity,
- support for distributed computing,
- support for multi-component and collaborative steering,
- utilisation method.

1.3.1 User Interfaces

The simplest interfaces available to users are *textual interfaces*, based either on console commands or on input read from textual files.

Some more sophisticated alternatives would be *visual programming languages* (VPLs), which provide the environment to design, compile, debug, etc., and even execute programs by manipulating graphically the components (“boxes and arrows” [96], e. g.).

Graphical user interfaces (GUIs) are still most common in modern computational steering applications. They typically consist of a visualisation window (possibly allowing for direct interaction) and a dialog-based interface, where a user can change simulation or view parameters through using sliders, text fields, boxes, etc.

Most complex and most immersive user interfaces are *virtual reality* (VR) *environments* such as cave automatic virtual environments (CAVEs), and flexible reconfigurable CAVEs – FRAVES. They consist of three to six walls in the virtual room, with different kinds of sensors, towards which high-resolution projectors are oriented. The actions of a user are detected by the sensors and the virtual environment responds to changes immediately.

1.3.2 Aim of Steering

According to [168, 127], it can be distinguished between human interactive, algorithmic, performance steering, and combinations of these.

In *human-interactive steering* a *user* modifies a simulation during the program run time in order to explore the computational model, while in *algorithmic steering* the *system* is instrumented to monitor intermediate results and/or statistics “online”, or store them in files.

Performance steering refers to migration and monitoring mechanisms, especially popular, and even essential, in distributed and grid services. It is used in load

balancing and scheduling, modelling and design, in interactive refinement of scientific visualisations, and performance optimisation of web servers and parallel I/O systems [113].

1.3.3 Type of Steering

There are also two so-called *types* of steering. One is steering and controlling a *program as an entity*, where all the interaction is done on the global, program level. The typical user actions which can be categorised here are: *pause*, *migrate*, *checkpoint*, *break*, *suspend*, *resume*, *kill*, etc.

In contrast to this, there is *application level steering*, where scientists can get much more insight from their simulations. They are able to modify and query the simulation data, including, for instance, some simulation internal state variables, inspecting as carefully the progress of the overall simulation, as the “behaviour" of the chosen individual parameters, the details of interest, the desired parts of the domain, etc.

1.3.4 Level of Interactivity

There are two mainstream approaches to steering in terms of how and when the changes are applied. So-called *“stop-and-go”* steering technique (known also as *“online monitoring”*) assumes explicit stopping or pausing the simulation, performing changes, and then explicitly starting anew from the point of interest. This mechanism allows not only for more time to analyse every action and the corresponding effect, but also more time to think about the next action.

In contrast to this, interactive steering can also be done *“on-the-fly”*, which is typically implemented via call-backs and similar mechanisms. It assumes real-time feedback from the running simulation, while experimenting with different simulation setups. The *intuitive* recognition of which change made by a user has lead to which effect is the prerequisite which is sometimes challenging to achieve. However, if this condition *is* met, interactive steering provides more immersive experience, i. e. situations where one can learn “by accident” and, thus, can come to an idea to examine a particular setting closer. The disadvantage is, however, having less time to think and plan the next action.

Both methods are challenging in that sense, that they require deep knowledge in wide range of disciplines. However, for very time and memory consuming simulations running remotely (even on massively parallel architectures), the latter method involves solving more complicated synchronisation issues among processes. Namely, it is necessary that the calculations remain consistent all the time, despite frequent changes. In this dissertation, the accent will be set exactly to this latter, more dynamic, hence, typically more ambitious approach.

1.3.5 Support for Distributed Computing

As the name already indicates, this classification is concerned with the distribution of components of a complete steerable application or a system. These may all be executed on a single (local) machine. Otherwise, either simulation or visualisation processes, or both, run remotely, distributed among (possibly) heterogeneous computing systems and communicating via a network (or even using Web or Grid services, e. g.).

1.3.6 Support for Multi-Component and Collaborative Steering

Running and steering *multiple simulations* simultaneously has certainly always been a point of interest. Sometimes there is also a need to attach *multiple visualisation clients*. The third sub-categorisation criteria is the number of the users, e. g. experts in different fields, that can attach to the application and steer it simultaneously. Multiple users certainly bring into play synchronisation issues and additional data sharing policies. However, support for multi-component and collaborative steering is precious in collaborative, interdisciplinary research.

1.3.7 Utilisation Method

As suggested in [127], it is clearly distinguished between the scenario where the existing applications can be incorporated into the steering environment, and the one where the environment itself is intended for a *development* of new applications. Looking from that perspective, there is a variety of steering tools developed over decades – from those where whole applications may be developed, such as *Problem Solving Environments* (PSEs), to the light-weight steering tools (i. e. *frameworks*, *toolsets*, *middleware*, etc.), which are intended mostly for a loose coupling to, or integration into, some existing code. The overview of the state-of-the-art tools is provided later, in Chapter 2.

Problem Solving Environments

As mentioned in the previous sections, one increasingly wants to put experts in the centre of the result analysis, in order to utilise the modern simulation possibilities in real-life problems. Therefore, it is essential to provide the tools where the complexity of individual components is hidden from those experts, so that they may only concentrate on the problem (i. e. a setup, processing, etc.) and the interpretation of the results.

Problem Solving Environments (PSEs) are popular tools that facilitate interactions with complex computational models, without requiring specialists to know their algorithmic, data, or visualisation structures [100]. "The environment must allow the scientist to focus on the science, and the visualisation expert to focus on understanding the data" [136]. In short, typically, these are user-friendly tools for guiding the numerically approximated solution to the governing system of equations for a certain problem.

These tools are intended to support solving the problem on all levels [94, 95]. First, they provide a sophisticated user interface as a tool for both software development – including debugging and tuning – and the interaction on the user level. Then, they provide at least basic modelling, computation, and visualisation components, which are preferably flexible and extendable. Finally, their role is to coordinate all the components. PSEs form typically a whole environment – an infrastructure where one may plug in different simulation and visualisation components. The rest, such as event handling, communication between the two, user interface components, etc. may well be part of the environment. It is possible to intervene during all phases of the model exploration – from the development to the performance tuning – and also during the execution of any of the aforementioned components.

Light-weight steering tools

Besides PSEs, there are various light-weight tools available for computational steering, such as: application frameworks, scripting languages, wrappers, middle-layers between simulation and visualisation, toolkits, library routines for instrumentation of a source code, etc.

1.4 Computational Steering in Applications

Experts in various fields would like to make as well-informed decisions as possible within their area of expertise. Some of them are neither educated how to develop simulation methods, nor how to develop the corresponding software. This is the reason why all of them – from medical doctors to architects – should tend to be supported with tools with sophisticated features, however, hidden complexity of simulation and visualisation methods, high-performance computing, networking, etc.

This section provides a broad overview of fields where computational steering is utilised. These are mostly, but not exclusively, scientific and engineering fields. At this introductory point, the intention of the author is purely to show how wide the application area actually is, or may grow and, hence, which experts already profit, or may profit, from such tools. The concrete software implementations themselves, in terms of the implementation concepts behind the existing frameworks, PSEs, libraries, etc. are described in Chapter 2.

1.4.1 Civil Engineering and Architecture

Computational steering has found its application already in quite a few scenarios in Civil Engineering. Those are concerned with engineering ventilation systems in rooms (Fig. 1.2, left), heating ventilation air-conditioning systems, flood simulations, etc. Only few of them are now briefly described.

Engineering Ventilation system in rooms, as proposed in [175], allows a user to explore indoor airflow while interactively adding, removing, or modifying objects

inside a virtual room. The modifications may refer to translation, rotation, scaling in each direction, etc. Heating Ventilation Air-Conditioning (HVAC) systems simulation described in [48] provides an environment where even multiple distributed researchers can collaborate working on a fluid simulator through inserting obstacles into a domain, relocating or removing them simultaneously and seeing the impact of these modifications on the fluid immediately. To analyse the indoor flow, an HVAC engineer will use a tool for computational fluid dynamics (CFD), while the interior designer might use a visualisation software with advanced rendering capabilities in order to perform an illumination analysis [49]. In the work of [138], single parameters can be modified or components added to the scene. The model considers the stationary heat balance of the human body, taking into account the statistically predicted reaction to ambient conditions, where both the metabolism and the clothing are considered.

1.4.2 Biomedical and Biology Applications

In biomedical applications desired properties of implantable devices cannot be explored experimentally (through the surgical interventions on humans). This is where numerical simulation tests are precious, to explore a wide range of design and placement possibilities or simply learn about the patient-specific abnormalities. To accurately model such a system, often a highly-detailed computational model must be used. Such a model can take hours or days of CPU time to compute results, such as, for instance, in Computational Cardiovascular Biomechanics applications.

One such example is simulation of Cardiopulmonary Bypass Surgery (CBS) proposed in [63]. It is an example of the simulation being finished out of the scope of the “interactive” rates and no run time steering is enabled yet. But, the motivation should come from the fact that out of more than 2 million procedures which are performed yearly, 2–13 % end up in some sort of a stroke (excluding other less serious complications). During this surgery, the deoxygenated blood is removed from the right side of the heart to a heart-lung-machine, where it is oxygenated, filtered, and pumped back to the circulation system, thereby bypassing the heart and the lungs. To provide the surgeons a bloodless environment of the heart to work on, the ascending aorta is clamped. Thereby, some particles may break loose and block the artery to the brain, or hypoperfusion may occur (not sufficient blood supply reaching all arteries in the brain). A complex interplay between the following factors plays a major role: patient-specific geometry, cannula site, orientation and tip design, clamp site and orientation, interaction between cannula and clamp, etc. Computational steering should then help a surgeon find some optimum state of the clamp and cannula sites which will have the least risk of causing the release of embolic particles and sustaining the most physiological flow conditions for the specific patient during this procedure.

Medical doctors would take a great advantage through exploring medical patient-specific simulations during run time and, thus, knowing what to look out for.

Quite a few of the applications, in fact, already provide support for interactive steering.

One such application, proposed in [153], serves for pre-operative planning of non-invasive vascular reconstruction, with a physician in the experimental loop. To achieve the best treatment, which is often not obvious for a surgeon due to a serious vascular disease, he[1] would like to preview the patient's condition and experiment by virtually adding a bypass for the patient specific scenario, i. e. a graft redirecting the blood around vascular blockages. In the interactive blood flow simulation within artery suggested in [174], one can add obstacles or virtual bypasses inside the simulated artery in order to narrow or completely obstruct the blood flow.

A pre-operative planning environment for joint replacement surgery is elaborated on in [178, 66, 67]. It is yet another application where patient-specific data is the crucial factor for choosing an optimal implant. An improper design, size, or position may lead to bone degeneration due to stress shielding, having unfortunately soon a new surgery as an outcome. The tool enables real-time surgeon's interplay with virtual models of bones and implants in sophisticated 3D visual environments (see Fig. 1.2, middle) providing an easily interpretable feedback in terms of bone stresses.

Within *SCIRun* PSE [1], the two following medical simulations can be run, as described thoroughly within the online tutorials [154, 155].

Myocardial ischemia is a disease characterised by reduced blood supply of the heart muscle. Symptoms may include characteristic chest pain on exertion and decreased exercise tolerance [2]. It is the most usual cause of death in most Western countries, and a major cause of hospital admissions [142]. Since early detection may lead to the prevention of further complications, scientists hope to detail what occurs in the border zones between the healthy and ischemic tissue layers by measuring many anatomic details and electrical measurements [100]. Since the simulation can be very time consuming, one would like to interact with the model, test different simulation properties and visualisation methods of the data of interest.

Defibrillation consists of delivering a dose of electrical energy to the affected heart with a device that terminates cardiac disorders and allows normal sinus rhythm to be reestablished by the body's natural pacemaker. Implantable Cardioverter Defibrillators (ICDs) are small, relatively common implantable electric devices, that provide an electric shock to treat fatal arrhythmias in cardiac patients [3]. Specialists need to be able to accurately both simulate and interpret how the assigned electric potentials in one part affect other parts of the domain, i. e. thorax and, thus, to choose an optimal, patient-specific device. Children, for instance, require more specialised ICD configuration than adult patients, due to their smaller size and often abnormal anatomy [58]. Optimal operation is

[1] Here and further in the text, for reasons of simplicity, a male form of a pronoun is used, however, it stands for both a male and a female person.

achieved only through the correct combination of voltage, placement, size, shape, and numbers of electrodes [136], where computational steering is advantageous.

Another light-weight version of steering has application to computational biology. One simulates the electrical behaviour of the heart – in particular, reentrant arrhythmia or irregular heartbeat. In the *gViz* [52] demonstrator, a number of heart simulations are run and the results monitored, each for a specific set of parameters which are of interest.

The *iPlant* project goal is to develop new tools, networks, and cyberinfrastructure that can connect plant biologists and bring together their data, in order to feed the growing population on Earth; a common *iPlant* API is developed and allows researchers with little programming experience to add common functionality to their plant biology projects.

1.4.3 Physics-Based Simulations

Physics-based computational steering applications described within this section are mostly related to computational fluid dynamics (CFD) and its application, astrophysics, heat conduction, electromagnetic wave modelling, etc.

Computational Fluid Dynamics (CFD)

FlowSim 2004.NET [4] is selected for this overview due to the fact that it gives a user the opportunity to interactively modify the flow domain. All the changes, for example to the geometry of the flow area, can be directly incorporated into the calculation.

In [38], one uses the Navier-Stokes equation to simulate the flow of an incompressible fluid on a complex surface. First, one sets some initial conditions, like the distribution of a coloured dye, which helps to visualise the flow. Then, one defines the boundary conditions, such as the sources of dye and the in-flow/out-flow of the fluid. Finally, the simulation runs and the results are visualised using different rendering techniques, chosen via user interface. A user can directly interact with the fluid under the mouse cursor, due to the interactive simulation rates.

In [35], the interactive simulation rates are achieved as well, using CUDA-based implementation. This 2D flow simulation allows effective interactive manipulation of a boundary, such as solid obstacles or velocity profiles.

The *CFDLIB* simulation, integrated into the *SCIRun* PSE[1], consists of three fluids of different density that interact inside the box. Scalar field of density, temperature and volume friction are produced for each type of a fluid. The pressure, temperature and velocity fields can be extracted from the running simulation and the particle sets can be extracted from a particle-based simulation. One can view datasets from previous time-steps using a slider.

In [59], a simulation of lid-driven cavity is shown. It is, namely, a box full of liquid and the top lid is moving from left to right with a certain velocity. The interactively visualised streamlines show how a particle would move if placed in the flow. The parameters are then being changed (viscosity of the fluid, velocity of the lid, and the time, where one can jump backward and forward). One simulation takes around 30 minutes to complete (in parallel) and the interpolation delivers an approximate solution in real time.

One of the “pioneers” of interactive steering of the simulation is a 3D turbulence model in Lake Erie [121].

Another application area in CFD is weather modelling. For instance, a meteorologist can control the wind direction [54] via Virtual Windtunnel Binocular Omni Orientation monitor.

Virtual environment techniques, just like in [175] turned out to be useful in visualising complex fluid flows. The flow can be investigated at any length scale. The time evolution of the flow can be sped up, slowed down, run backwards or stopped completely for detailed examination.

Similarly, in atmospheric modelling of *Distributed laboratories* [141], assimilated wind fields derived from satellite observational data are used for its transport calculations, and known chemical concentrations are derived from observational data for its chemistry calculations. Models like this are important tools for answering questions regarding exchange mechanism between different layers of atmosphere or the distribution of species such as ozone. Steering is accomplished by positioning latitudinal and longitudinal planes, sizing and moving the cube to intersect the plane, and then entering a desired concentration value so as to apply it to all grid points in the cube.

A wide scope of other physics-based applications ranges from thermodynamics, such as heat diffusion simulation [167], to astrophysics, etc., as described in the following, last, example within this category.

The *Enzo* simulation software is incredibly flexible, and can be used to simulate a wide range of cosmological situations with the available physics packages [5]. It is mostly online monitoring, however on very large scales for the present moment. The model represents a volume of the Universe about 800 million light years on a side and used a 5123 top level mesh with 7 levels of adaptive mesh refinement. It requires the memory and processing power of almost 94 000 cores of the NICS Cray XT5, Kraken. Each re-start/checkpoint file is 30 TB in size [97].

1.4.4 Pollution Modelling

Pollution modelling is an another area of application. An environment disaster scenario is considered in [52]. The dispersion of a dangerous environment pollutant is simulated, in order to ensure efficient evacuation. Evacuation planners can experiment with different wind directions via arrow widgets of a visualisation front-end.

Another example is smog prediction over Europe [126]. The simulation calculates the smog dispersion over four layers according to the numerous input parameters concerning emission fields, meteorological and geographical parameters. The governing equations of the model are a set of partial differential equations that determine the advection diffusion emission, wet and dry deposition, fumigation, and chemical reactions. In this interface, the user can examine the different atmospheric layers and steer the simulation by manipulating pollution emission sources.

Atmospheric diffusion is another representative application example. A simulation of atmospheric dispersion from a power station plume is described in [136]. The photo-chemical reaction of NO_x with polluted air leads to the generation of ozone at large distances downwind from the source. The user interface allows the user to set up initial conditions: the initial position of the pollution source, the velocity of the flow, the level of adaptive mesh refinement in the regions close to the pollution source, etc. The main area of mesh refinement is along the plume edges close to the chimney. Using an adaptive mesh, the plume edges can be clearly seen and areas of high concentrations identified. During the execution time, a user can change the level of adaptive mesh refinement.

1.4.5 Nuclear Engineering

The first example in this field is the *AGENT* simulation framework for modelling reactor physics in configurations common to standard research reactors, current power reactors, and any future reactor designs. *AGENT* solves the 3D Boltzmann transport equation using the method of characteristics (MOC). This method of simulation requires a large number of parameters to be configured by the user – a variety of meshing parameters, as well as a set of convergence variables. Mesh and resolution parameters include, for example, the number of rays representing neutron tracks of motion and the number of boundary edges along each side of the reactor core assemblies. Improper settings of these and other parameters can lead to slower and worse convergence of neutron transport solution and, thus, dissipated computational resources or memory. However, the proper values are not easily understood in many cases, and hence the user is required to complete a detailed “survey” toward the best estimate solution. This unnecessarily exacerbates the computational cost of simulating particular phenomena. To resolve this, a computational steering approach is employed. In some other reactor modelling environments, such as at the one developed at the University of Illinois [99], “only” online monitoring is used thus far (to the best knowledge of the author).

1.4.6 Steered Molecular Dynamics

Molecular dynamics is also an example of a field where computational steering plays an important role, as presented in [64]. The described tool serves for creating and examining new molecules – sometimes called Steered Molecular Dynamics (SMD) [24]. Here, the user can express external forces to help the system overcome energy barriers or he can search for likely geometric configurations

in docking problems [118, 119]. Another scenario is the *Distributed Laboratories* [141] project, which makes possible the performance optimisation, such as changing decomposition geometries in response to changes in physical systems or shifting the boundaries of spatial decompositions for dynamic load balancing among multiple processes. The aim of the simulation is calculating intra- and inter-molecular forces based on information from neighbouring particles in order to apply calculated forces to yield a new particle position.

1.4.7 Manufacturing

In [64] the advantages of numerical simulations for a semiconductor diode laser are described, subject to optical feedback, where the result has infinite degrees of freedom [64]. The user can interactively set the values of selected parameters using sliders. He can choose any of the (visualised) fixed points to start the simulation from, or can experiment with different parameter values.

In wood science applications, research into the time of press closure in the hot compression process has always been a mystery. It only lasts for 15 to 60 seconds, but much of the final panel characteristics are believed to be determined during this time. There has been a lot of experimental work to determine the influence of press closure rate on panel properties. The challenge is to determine when the key events occur. A scientist can use the *WBCSim* [151] computational steering feature to study this issue, instead of physically testing a panel after it has completed the entire press cycle. Here, the parameters related to the steering setup, material, initial condition, boundary condition, adhesive cure, and press schedule can be specified. The feature provides intermediate results and options to steer the simulation during the run. The model is defined by specifying material transport, boundary transport, and compression properties.

TENT [75] is used in analysis of turbo components in gas turbines, for the development of virtual automobile prototypes, the simulation of static aeroelastics of an aircraft and the simulation of combustion chambers.

More examples of interactive simulation steering of manufacturing systems can be found in [98], automotive industry and, to be more specific, vehicle forward lightning – optimising for visibility and comfort, where virtual lightening laboratory from Hella KG [45] has been involved.

1.4.8 Other Application Areas

Indoor thermal comfort is becoming increasingly important in the industrial environment. Relevant areas of application in this scope, according to [166, 138] include manufacturing, such as the automotive industry, the aircraft industry, the railway/coach industry, as well as Civil Engineering – the HVAC-building sector, for instance.

Acoustic wave propagation has been studied by interactive adjustment of input parameters, such as the seismic shot location, intensity, etc. [21, 77]. A user

may either get a coarse overview of the whole computational domain, or a finer, detailed view of a particular area of interest, i. e. the shock-wave reaction.

Simulation of the sailing characteristics of an East-Indiaman is described in [126]. The simulation calculates the new state of the ship at each time step. The new state is derived from the prior state and the wind conditions (the direction and the force), the wave spectrum of the see, the position, orientation and surface area of sails, the orientation of the rudder, etc. The user can literally *steer* the ship – sails can be set, the sail area can be enlarged or reduced to adjust to different wind forces, the sails can be aimed to different angles to adjust to different wind directions and ship courses; the user can use the rudder and the sails to change the course of the ship; the wind force and direction can also be changed.

The next example is *C-SAFE* (Fig. 1.2, right) [6]. Its ultimate goal is to simulate fires involving a diverse range of accident scenarios including multiple high-energy devices, complex building/surroundings geometries and many fuel sources. However, the initial efforts focus on the computation of three scenarios: rapid heating of a container with conventional explosives in a pool fire (e. g., a missile involved in an intense jet-fuel fire after an airplane crash), impact and ignition of a container with subsequent explosion and fire-spread (e. g., shelling of a mine storage building by terrorists) and heterogeneous fire containing a high energy device (e. g., ignition of a containment building in a missile storage area).

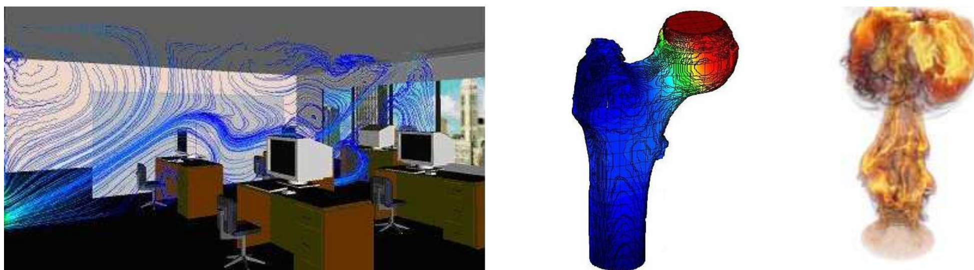


Figure 1.2: Diverse application fields for interactive computing – left: air flow simulation in an office room, middle: bone simulation (source: Chair for Computation in Engineering, TU München), right: fire simulation (source: C-SAFE, University of Utah).

Many present computer games use concepts closely related to computational steering. Just to name one, an interactive *RoboCup* (Robot Soccer) [64] is an interactive environment which lets teams play a soccer match together, using either real robots or simulated players [64]. People at different locations on Earth can play along with a running RoboCup simulation in a natural and intuitive way – each in his own CAVE.

Other applications may include traffic, crowd, flood simulation and – last but not least in this overview – educational software. This software helps not only teachers evaluate the progress of students programming activity based on the collected statistics (e. g the speed of typing, hopefully indicating the program-

ming proficiency of the students) [161], but also supports students by tools for education and learning in all aforementioned areas of science and engineering, such as nuclear engineering, physics, mechanics, civil engineering, etc.

1.5 Real-Time Interactive Computing Environment

In order to provide an environment for experts to do analysis in the existing, widest imaginable, range of fields (only some of which are mentioned in the previous section), "scientists attempt to use as much resolution as they have memory and patience for" [136]. As it can be concluded from the previous examples, multi-scale, multi-dimensional, and multi-physics simulations are nowadays the rule rather then the exception. This is also where the motivation comes from to explore *interactively* the underlying complex models. Preparing, running, and postprocessing within a large-scale simulation takes away minutes to hours, or even days of a researcher's time. Yet, at least several updates per *second* would be desirable for interactive computational steering. What could be done then?

Numerous aspects have to be considered in order to create an advanced environment, with real-time computational steering enabled for a wide range of applications, in particular large-scale, multidimensional, multiphysics simulations. Let us optimistically assume that powerful hardware is available, as well as an efficient application basis (allowing for utilising that hardware). So as to fully develop such an environment, it is still required to get acquainted with numerics, high-performance computing, network transport protocols, and visualisation, as discussed in more detail in Chapter 2. However, even this is not necessarily solving all the open questions.

The situation gets more challenging if the simulation kernel is executed remotely on a supercomputer cluster, in contrast to where graphical user interface and visualisation are done typically, i. e. on a local machine. The simulation may be required to run in an interactive instead of a batch mode, which is not always supported by supercomputer systems [76].

Finally, it is also important to have systematic, patterned, consistent approach to the design, development, and maintenance of software.

1.6 Task Definition and Approach

Current computational steering problem solving environments, libraries, and frameworks significantly help engineers run some specific simulation codes in an interactive mode. Nevertheless, when it comes to real-time response of the simulation to this interaction – namely, keeping the aforementioned relation between a user's change and its effect intuitive or at least observable – these environments are still limited in their possible application. In other words, they are typically restricted to just one, or very few scenarios for which they perform exactly as desired. Moreover, they often entail heavy code changes in order to be coupled to existing codes.

Therefore, the goal of this research is to implement an integration framework with the following properties:

- applicable to different engineering applications,
- minimal invasive – modification of the original application code necessary are minimal,
- supports multithreading, distributed parallel simulations, as well as visualisation on-the-fly,
- supports different hierarchy-based/adaptive simulation methods,
- provides instant response of the simulation model to changes (keeps the connection between a cause and the corresponding effect intuitive),
- provides fast transfer of a simulation result,
- can be compiled using different compilers,
- is portable to different operating systems (OSs).

To avoid any kind of ambiguities of the terms and in the same time stress the generic character of the approach described within this work, it will be denoted *interactive computing* further on in the text and, hence, its implementation with the listed properties – an *interactive computing integration framework* or sometimes just *framework*.

Yet, the author would like to point out that it is mostly aimed to serve numerical engineering applications. It is inspired by various engineering disciplines, some of which are mentioned later, in Chapter 4, and developed in engineering-dominated academic surrounding, where the motivation for introducing a relatively generic, easy to integrate concept has originally come from.

To sum up, in the modern scientific computing community, the author reasons that it should not, without any consideration, be claimed unimaginable to interactively steer also long-running, resource intensive simulations, providing at least some feedback about the effect of modifications in real-time. *Long-running*, in this context, may mean anything which is not real-time – in the range from a second to a day, or longer. *Resource-intensive* means – in the context of this work – requiring full utilisation of the available hardware resources: machine memory, network bandwidth, etc. Most of the aspects mentioned in Section 1.5 are considered. Considering these aspects is essential in order to fulfill the aforementioned vision in the future.

Thus, different interactive steering concepts are thoroughly discussed throughout this dissertation, some of them applied and evaluated. In addition to the implementation-independent option to switch to more powerful hardware, good utilisation of that hardware is assumed within all application components and

layers. Usage of more efficient algorithms and data structures is advised as an optimal basis for integration of interactive computing concepts. Some of the sophisticated domain decomposition methods are portrayed, allowing not only for new designs of faster solvers, i. e. with lower computational complexity, but also sophisticated parallelisation strategies. Reduced order models – or simply reduced accuracy requirements – are suggested while an application is running in an interactive mode (if they may be appropriate for the particular application).

The author considers this a helpful contribution for proposing a direction in which science and engineering might go when it comes to implementation of interactive computing tools. For realistic problems, where typically large data-sets as well as high computational complexity requirements have to be handled efficiently, the potentially detected performance limits could often be overcome via sophisticated simulation and visualisation techniques (as a basis) and good performance of the overall interactive environment. The latter in particular could/should be provided via user-friendly interfaces, with all the implementation details "hidden" from an end-user. The aim is to support research in various fields and *encourage* experts' demand for such tools both in industry and in other application areas.

1.7 Guide Through Chapters

This chapter has introduced the motivation for doing interactive computational steering and its numerous challenges. The taxonomy of the tools enabling interactive computational steering helps to understand their differences, which are often colossal. The chapter provides a broad overview of application fields, without any implementation details of the computational steering environments supporting the interactive process. The aims of minimalistic integration frameworks for a wide scope of engineering applications – developed as a goal of this thesis – are described, together with the concept of "interactive computing", the way the author sees it.

Chapter 2 describes in more detail the challenges in building a real-time interactive computational steering environment. The chain consists of (1) efficient simulation methods based on approximation and sophisticated solvers, which exploit different domain decomposition methods and hierarchical approaches, (2) advanced parallelisation strategies, (3) fast transfer of simulation result, (4) editing and visualisation of this result. The actual implementation of the (classified) state-of-the-art tools supporting interactive steering is discussed at *this* point. The difference of the concept introduced in this work is briefly presented.

Chapter 3 provides a detailed picture of the concept of the developed framework, as well as the implementation specifications. The concept of *signals* is introduced, due to its relevance for the framework implementation. Examples illustrate the implementation details and challenges not only for sequential application codes but also for parallel ones – multithreaded, distributed parallel and "hybrid". Preliminary results are shown in regard to the overhead of the frame-

work and, moreover, in regard to the minimal effort which a user has to invest in order to enable this pattern in his own application.

Chapter 4 presents the experience gained by the integration of the interactive computing concept into different engineering scenarios, as well as the achieved results. The first application scenario is a 2D simulation of the temperature conduction, where heat sources, boundaries of the domain, etc. can be interactively modified. The second one is a neutron transport simulation developed at the University of Utah's Nuclear Engineering Program, which has served as the first Fortran test case for the framework. Next is the sophisticated Problem Solving Environment SCIRun developed at the Scientific Computing and Imaging (SCI) Institute, University of Utah. Due to its flexibility, modularity, and dataflow based design, it is used in different applications at the SCI Institute and wider, thus, has provided an excellent basis for interactive computational steering and the test integration of the proposed framework. The fourth, final one, is a tool for pre-operative planning of hip-joint surgery, done as a collaborative project of the Chair for Computation in Engineering and Computer Graphics and Visualization at Technische Universität München. For each individual scenario, the effort to integrate the framework is briefly discussed from the user's point of view. The overhead of the framework is analysed in terms of the execution time.

Chapter 5 summarises all the conclusions from the previous chapters and announces the outlook for future work.

Appendix presents the exact practical changes which were made in order to integrate the proposed framework into the heat conduction application code.

Chapter 2

Interactive Computing

This chapter introduces, above all, the minimal prerequisites for building up an efficient and scalable, high-performance interactive computing environment. Common central requirements in interactive computing are to support development and execution and, thus, to design corresponding components. The development (i. e. front-end) part should involve a sophisticated user interface, which guides the user in building a solution to his problem, and at least one visualisation widget, which may itself also be interactive. On the back-end often executes a time- and memory-consuming simulation kernel, either on a local machine or on a high-performance cluster (Fig. 2.1).

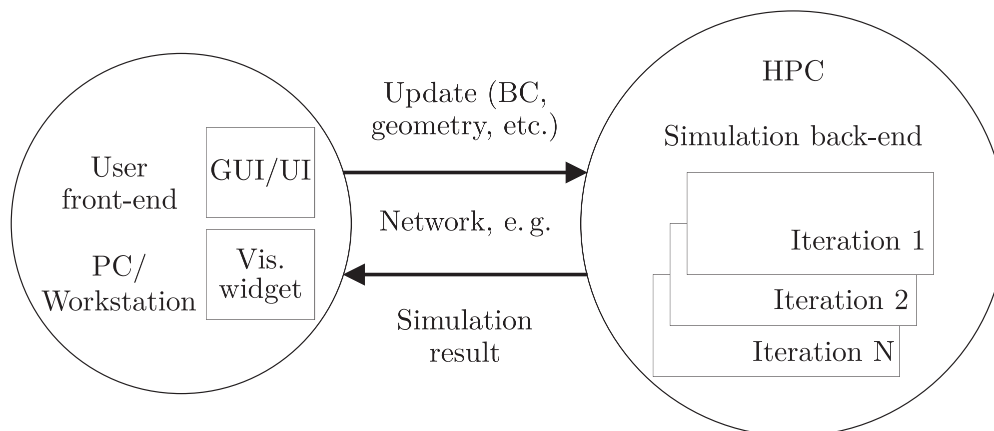


Figure 2.1: A large-scale simulation is executed on the simulation back-end, while on the visualisation front-end, the user guides the solution via a user interface (changing boundary conditions, geometry, etc.).

The state-of-the-art implementation of such tools will be briefly presented and classified using the taxonomy defined in Chapter 1. The awareness of concepts already utilised worldwide and the features they support, as well as the experience gained within our research group over the last decade, have contributed to the success of establishing a new, rather generic, interactive computing concept.

Thus, this chapter discusses the barriers preventing real-time update rates as a response to user interaction especially for large-scale problems, as well as the optimal application basis for the integration of this concept. There is no restriction imposed on a potential application scenario so far. However, in conjunction with an optimal application basis, the integration of this new concept, i. e. framework, involving only minor code changes, may result in complete, efficient interactive computing environments, at least for many familiar standard-class optimised application cases. Nevertheless, in addition to the numerous challenges already addressed, the growing number of application scenarios will certainly always present new challenges. While the solutions to these challenges may increase the complexity, they may also contribute to a more robust, feature-rich framework.

2.1 Challenges in Building a Scalable Interactive Computing Environment

Solving small-scale problems interactively has already been thoroughly investigated in the field. For building up an environment with real-time interactive steering enabled for large-scale problems, the developers still have to bring together a variety of components, even assuming state-of-the-art hardware is available (i. e. high performance CPU/GPU clusters, high-bandwidth network, etc.):

- an efficient (rather parallelised) simulation, running in a high-performance computing (HPC) environment,
- a possibility for a user to interrupt the simulation at any point (to have it re-executed with the updated settings),
- hierarchical concepts (e. g., computing with different approximation-related parameters) providing less accurate, albeit quick intermediate feedback from the simulation,
- fast transfer of simulation results, using methods for selection or compression of data, via high-bandwidth network connections,
- intuitive and real-time visualisation of this result.

These are the minimal prerequisites, all of which have to be taken into account to provide the desired response of a large-scale simulation to user interaction.

This chapter gives an overview of the involved aspects, with the corresponding state-of-the art hardware where appropriate, to describe the optimal basis for integration of the interactive computing framework. The application bases already given and tested with the framework are also discussed. The framework is not held responsible to make the provided application scenario – e. g., a simulation method – more efficient. On the contrary, we strive for a minimally-invasive integration. However, potential users should keep in mind that the initial application code, once it completely fulfills the requirements suggested in the previous

paragraph, could significantly contribute to the interactivity rate of the overall system, i. e. application software "symbiosis" with the integrated framework.

Especially where a simulation of complex, realistic phenomena is concerned, without efficient simulation methods, interactive computing concepts alone could never lead to acceptable update rates as a response to user modifications. *Acceptable update rates*, within this work, means the rates which keep intuitive for a user the connection between his interaction and the corresponding outcome effect.

Namely, only so-called *real-time simulations* (irrespective of how this "real-time" property has been achieved) allow instant transfer and visualisation of results as a response to user interaction, independent of the realisation of a coupling of the two components.

Contrarily, there are cases when *middle-to-large-scale* problems are not amenable for parallelisation, or even those with a "decent" level of concurrency of the tasks involved, due to inadequate parallelisation schemes, or any other reason, still remain out of the scope of real-time simulation rates. It may be the case that neither CPU/GPU power of PCs/workstations, nor of supercomputer clusters, are sufficient to compensate for a poor parallelisation strategy, or any other software property due to which only some of the hardware capacity can actually be exploited. An effort to rethink the algorithms and rewrite the codes might pay off in this case. When solving a system of equations, for example, strategically used spatial decomposition methods may help and allow not only for faster direct solvers, but also for scheduling and load balancing methods needed for efficient parallelisation. This is the case with the nested dissection algorithm described later.

For *large-scale problems*, on the other hand, even when a simulation scales well on powerful hardware, it is often rather questionable whether constructive (or at least intermediate) feedback from the simulation back-end could be provided in even close-to-real-time. To that case, simulation authors can turn to:

- adaptive approximation methods, or multi-level, or reduced order methods, which can be physics based, or projection based (to a lower dimensional space which captures most of the information of the original system),
- sophisticated iterative solvers or even hierarchical solvers – hierarchical preconditioners and efficient algebraic solver methods, such as multigrid (if there *is* a system of equations to be solved, keeping in mind a wide scope of possible simulation scenarios),
- etc.

Therefore, a closer overview of the common simulation methods and parallelisation techniques which may be applied is provided – especially those which are directly related to the application and test cases with the framework integrated.

2.2 Efficient Simulation Basis

For the scientific simulation of phenomena within an interactive computing environment, an *efficient* modelling of a desired physical problem domain is required, as well as appropriate boundary conditions.

Based on the application scenario, appropriate numerical approximations of the governing system of equations of a particular phenomenon (FEM, FD, FVM) should be chosen, which provide fast computation of the solution. Thereby, sometimes an opportunity should be regarded and used to pre-compute potentially needed values and simply load them within an interactive computing loop, instead of computing them during the execution time.

2.2.1 Approximation

Approximation starts with describing the idealised physical problem with a mathematical model. Then, a discretisation step has to be done – often both *space* (related to the domain geometry) and time, and so-called *equation* discretisation.

Space discretisation means representing the continuous domain with a finite number of points or elements, to which discretisation entities are associated (such as shape functions in a Finite Element Method) and where the numerical values of simulation-specific variables have to be calculated. Based on the discretisation points a mesh is generated. Efficient and sometimes adaptive mesh generation is essential when it comes to any user updates which involve the re-meshing of the domain. Alternatively, for complex geometries, methods with an embedded domain approach, such as the Finite Cell Method (FCM) [72], are likely to be profited from, since mesh generation need only be done once. More remarks on the convenience of this approach when it comes to interactive steering are found with regard to the *Bone* application scenario in Chapter 4.

The equations are discretised by transforming the differential or integral equations into discrete algebraic operations involving the values of the unknowns related to the mesh points. As mentioned before, the resulting problem can be then an algebraic linear or nonlinear system of equations for the mesh-related unknowns.

The Finite Difference Method

The Finite Difference Method (FDM) is based on the properties of Taylor expansions and on the straightforward application of the definition of derivatives. It requires a grid to be set up in a structured way. The order of accuracy of FD formulas for a continuous function is evaluated by Taylor expansion for the function around some point x (in some small but finite Δx surrounding). Finite Difference formulas are said to be of the first, second, etc. *order*, depending on the order of accuracy, i. e. the power of Δx with which the dominant term in the local truncation error of the Taylor expansion tends to zero as Δx tends to zero. For every derivative, an infinite number of FD formulas can be derived,

depending on the number of mesh points involved and on the required order of accuracy. Theoretically, both can be arbitrarily high. Depending on the position of the points involved in the FD formula for a particular point, there are backward, forward, central, or mixed FD schemes. Not all these schemes, nor their derivation (which is straightforward; see [84]) will be discussed here. We only discuss the already derived (and utilised) formulas for particular application scenarios relevant in this work.

Difference schemes for the Laplace operator $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ are used in several application and test examples (Chapters 3 and 4) and are therefore introduced here. The Laplace equation for a (continuous) function u

$$\Delta u = 0, \tag{2.1}$$

and its generalisation, the Poisson equation

$$\Delta u = f, \tag{2.2}$$

play an important role in CFD, as well as in simple models such as heat conduction or potential flows (as will be seen later in the *2D heat conduction* and the *SCIRun* application scenarios). As the Laplace equation is typical for diffusion phenomena, which is of isotropic nature, i. e. does not distinguish between upstream and downstream directions, it has to be discretised with central differences, in order for the discretisation to be consistent with the physics it simulates. This is a crucial element in the selection of an adequate numerical scheme, among all the possible options [84]. Application of second-order central differences in both directions leads to the well-known five-point difference operator

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f_{i,j} \tag{2.3}$$

or for $\Delta x = \Delta y$

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = f_{i,j} \cdot \Delta x^2. \tag{2.4}$$

Here, f is a discrete "source" function and *(i, j)* an ordered pair of indices determining position of the discrete values of u and f associated to the grid (their x and y coordinates). This is the most widely applied scheme of second order accuracy for the Laplace operator [84].

The Finite Volume Method

In contrast to FDM, where the discretised space is considered a set of points, in the Finite Volume Method (FVM) a discretised space is formed by a set of small cells, one cell being associated with one mesh point. FVM associates a local finite volume (the "control volume") to each mesh point. It is a widely used technique in CFD, however, it is not discussed for this work, since it is utilised neither in our exemplary nor our application test cases.

The Finite Element Method

The Finite Element Method (FEM) [160, 19, 145, 86] is a well-established and widely-used numerical approximation scheme, especially in complex structural problems, where it originated. The first step, in general, is to decompose the given spatial domain into a finite number of elements, resulting, typically, in tri- or quad-elements in 2D or in tetrahedral or hexahedral elements in 3D, respectively. For each of the elements, so-called *basis*, *ansatz* or *shape* functions (either linear or higher-order) are defined. Then, representations of the unknowns based on linear combinations of the ansatz or shape functions associated to that element are formulated. Next, an integral formulation of the equations to be solved is defined for each element of the discretised space, i. e. the local problem. The global system of equations is assembled from all the individual elements.

To illustrate the concept on a practical example, in structural mechanics one wants to have the response of a structure under load. FEM can be applied to obtain an overview of the stresses and strains. These can be computed if the deformation under loading of the structure is known. It will be illustrated later on the *Bone* application test case, where the behaviour of a thigh bone under load – analogous to loading e. g. while a human is walking, climbing the stairs, etc. – can be simulated and interactively explored.

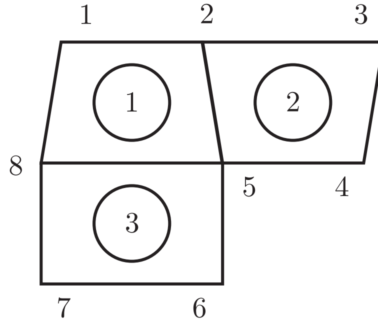
The behaviour of a classical type of elements, the so-called *Lagrange elements*, can be described in terms of the loads and responses at discrete nodes. For each element of a certain type there is a small matrix, the *element stiffness matrix*, which relates a vector of unknowns – *nodal displacements* – to a vector of applied forces at these nodes, the *load vector*. The elements of the matrix describing the elastic deformation under load are computed by integrals over the partial derivatives of the shape functions and the material parameters of the element. Once we calculate elements' stiffness matrices and load vectors, they are all assembled into one large matrix: the *global stiffness matrix* for the whole system, and the *global load vector*, respectively, due to the superposition principle (Fig. 2.2). Thus, the algebraic system of equations describing the behaviour of the whole system is

$$K \cdot u = d, \tag{2.5}$$

with K the system stiffness matrix, u the nodal displacements, and d the accumulated forces.

There are many variants of FEM, however, the mainstream approaches are:

- h–method: to subdivide the domain mesh (h is customarily the diameter of the largest element in the mesh),
- p–method [145, 73, 159]: instead of making h smaller, one increases the degree of the polynomials used in the basis function, p. Most p-version implementations use hierarchical shape functions and not Lagrange shape functions, with significant advantages concerning, for example, the condi-



$$(1)\ K_1 \cdot d_1 = f_1$$
$$(2)\ K_2 \cdot d_2 = f_2$$
$$(3)\ K_3 \cdot d_3 = f_3$$

$$K_{global} = K_1 \oplus K_2 \oplus K_3$$
$$f_{global} = f_1 \oplus f_2 \oplus f_3$$

Figure 2.2: The behaviour of a bilinear Lagrange element in FEM is described in terms of the loads and responses at discrete nodes 1,. . . 8. For each element, a small matrix, *element stiffness matrix*, K_i, $i \in 1, 2, 3$ is calculated, relating a vector of unknowns – *nodal displacements* d_i – to a vector of applied forces at these nodes, *load vector* f_i. The global system of equations $K_{global} \cdot d = f_{global}$ is gained by superposition from all the individual elements.

tion number of the stiffness matrix, yet loosing the property that degrees of freedom can directly be associated to nodal displacements.

- *hp*–method [71]: a method combining these two refinement types,
- *rp*–method: to adjust the FEM mesh to the shape of the elastic-plastic interface [133], etc.

The convenience of these adaptive approaches in one interactive computing environment will be thoroughly described in Chapter 3.

Method of Characteristics

Another method in use is the computational **Method of Characteristics (MOC)**. To briefly demonstrate the idea, a simple concrete example from [139] is provided. Let us consider a substance flowing in a region of interest $[x_1, x_2]$ in x-direction, $u(x, t)$ denoting its density ($[quantity] \cdot [volume]^{-1}$) as a function of position x and time t, and let $\phi(x, t)$ denote the flux ($[quantity] \cdot [time]^{-1} \cdot [area]^{-1}$), where density and flux variations in the y and z directions are assumed to be negligible. Velocity is denoted $c(x, t)$, and $f(x, t, u)$ the source term at which density increases by any process other than flux. Thus, the conservation law for a substance (e. g. mass or energy) in this 1D space

$$u_t + \phi_x = f \tag{2.6}$$

must hold across the whole domain. If we know the velocity $c(x, t)$ ($[length] \cdot [time]^{-1}$) then the flux is $\phi = cu$. By the trivial substitution, we get the transport equation

$$u_t + (cu)_x = f, \tag{2.7}$$

where one seeks the function $u(x,t)$ that satisfies this equation, and satisfies $u(x,0) = u_0(x)$ for some given initial density profile u_0.

After straightforward transformations of the transport equation, one can assign a geometric interpretation to it (see Fig. 2.3): we seek a surface $u(x,t)$ whose directional derivative in the direction of the vector field $\begin{bmatrix} c(x,t) \\ 1 \end{bmatrix}$ is $f - u \cdot c_x$. This geometric interpretation is the basis for the solution method. Curves $x = x^*(t)$ in the (x,t) plane that are tangential to the vector field $(c(x,t),1)$ are called *characteristic curves.* The characteristic curve that goes through the point $(x,t) = (k,0)$ is the graph of the function x^* that satisfies the ODE

$$\begin{aligned} \frac{dx^*}{dt} &= c(x^*,t), \\ x^*(0) &= k. \end{aligned} \tag{2.8}$$

Denoting the value of u along a characteristic curve by $u^*(t) = u(x^*(t),t)$ results in

$$\frac{d}{dt}u^* = \frac{\partial u}{\partial x}\frac{dx^*}{dt} + \frac{\partial u}{\partial t} = cu_x + u_t = g. \tag{2.9}$$

That is, characteristic curves are paths along which the differential operator of the transport equation reduces to a total derivative. They define equation discretisation for a flow simulation – the approximate paths along which particles move – supplemental to the geometry mesh discretisation.

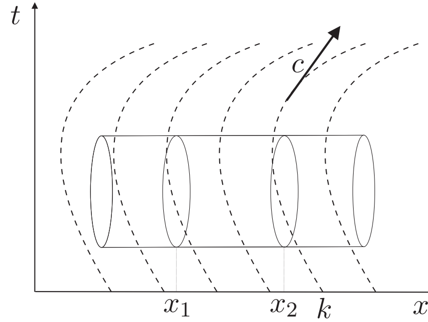


Figure 2.3: MOC: along characteristic curves the partial derivative in the transport equation reduces to a total derivative.

Among the application scenarios described in this thesis, the *2D heat conduction* test case uses central differences as an approximation method, *SCIRun* simulations are based on *h*-FEM, and the *Bone* project on a variant of *p*-FEM with the embedded domain approach, which, as already mentioned, can be extremely convenient for interactive computing since an expensive re-meshing step does not have to be repeated for each new configuration. The *AGENT* application uses MOC to simulate neutron transport (e. g., scalar fluxes) within a reactor core. Most of these application scenarios may use as well some kind of adaptivity or

hierarchy, which is discussed in Chapter 3 in the context of the integration of the framework. Namely, the intention is to show how these hierarchies can be exploited to accelerate the convergence of the solutions, while maintaining the desired accuracy, and how the framework fits into that concept.

2.2.2 Solvers

At this point, it is convenient to mention the three essential error sources in numerical simulation: the mathematical model error together with the input data error; approximation error – when operators or input parameters are replaced by approximations (e. g. a differential with a difference, a function with a polynomial); and round-off error which occurs during the calculation.

Once a resulting system of equations is obtained via one of the aforementioned approximation methods, it is essential that the solution method for it is effective as well. Depending on problem size and other properties, direct or iterative solvers, with or without preconditioning, are employed, or the popular hierarchical methods, such as multigrid (both as a stand-alone solver, and as preconditioner), or even their combinations.

As mentioned at several points before, the outcome of an approximation method is an algebraic system for the unknowns associated with a series of mesh points. Often it is simplified by discretisation into a large *linear* system, as it is the case with all the application cases presented within this thesis. This system can be very large. Two large families of solvers for linear algebraic systems are *direct* and *iterative* methods.

If this system is small enough to fit into memory, methods exist to find the direct solution efficiently. An overview of these methods can be found in [69]. Direct methods are based on a finite number of arithmetic operations leading to the exact solution of a linear algebraic system [84]. However, *Cholesky decomposition* and *Gaussian elimination* have a cost of $\mathrm{O}(N^3)$, where N is the number of variables. This is generally considered too costly in both memory and computational time. Sparse direct methods such as nested dissection cost $\mathrm{O}(N^{3/2})$ [114, 60], which is still high for large N.

However, it might not be necessary to solve the linear systems exactly. Depending on the application, approximate solutions, or even coarse approximations, can be sufficient.

Iterative methods perform operations on the matrix elements of the algebraic system in an iterative way. Their goal is to approach the exact solution, with some accuracy tolerance, in a finite, hopefully small number of iterations [84]. Most of these methods are not described in this dissertation, except for: (1) basics of the methods demonstrating hierarchical state-of-the-art solvers, essential for efficient simulation, (2) a few examples which are directly used in our application or test scenarios, and are also helpful for understanding the parallelisation strategies relevant for this work.

Simple iterative methods such as *Jacobi* and *Gauss-Seidel* have $O(N^2)$ cost per iteration, however, the number of iterations required depends on the condition number of the system matrix and on the desired level of accuracy. When the condition numbers are very large, these solvers might take too many iterations until solutions converge. This is the case, e. g., for some structural mechanics simulations in which p-FEM methods are utilised, one of which is the *Bone* application scenario.

Even in this case, iterative methods such as *Jacobi* and *Gauss-Seidel*, could be effective as so-called *smoothers*. They could serve to smoothen high-frequency components of the correction in small number of iterations. Hence they are used in multigrid methods, described later in the text, in conjunction with a hierarchy of grids. Since stand-alone *Gauss-Seidel* and *Jacobi* are used in a few framework application test cases, they are illustrated and discussed further in the text.

Examples of point Jacobi and point Gauss-Seidel method for the Poisson equation

In order to solve the Poisson equation (Eq. 2.2), an initial approximation (guess) of the vector u is defined. Then, an attempt to improve this approximation is made by sweeping through the mesh of $I_{max} \times J_{max}$ point-wise, starting at the point $(i, j) = (1, 1)$, i. e. with lowest mesh indexes in this example, traversing the mesh row- or column-wise. If we indicate by $u^n_{i,j}$ (or u^n_I) the assumed approximation (n will be an iteration index), the corrected approximation $u^{n+1}_{i,j}$ (or u^{n+1}_I) can be obtained according to the Gauss-Seidel scheme by

$$u^{n+1}(i,j) = \frac{1}{4}(u^n(i+1,j) + u^{n+1}(i-1,j) + u^n(i,j+1) + u^{n+1}(i,j-1)) - \frac{1}{4}f^n(i,j) \cdot \Delta x^2, \tag{2.10}$$

where $i \in [2, I_{max} - 1]$ and $j \in [2, J_{max} - 1]$. It can be observed that the points $(i, j-1)$ and $(i-1, j)$ have already been updated at iteration $(n+1)$ when $u_{i,j}$ is calculated. The new values in the estimation of $u^{n+1}_{i,j}$ are used as soon as they have been calculated. It can be expected thereby to obtain a higher convergence rate since the influence of a perturbation on u^n is transmitted more rapidly.

A program to solve the Poisson equation with Gauss-Seidel is illustrated in Algorithm 1.

The Jacobi method is defined by the iterative scheme

$$u^{n+1}(i,j) = \frac{1}{4}(u^n(i+1,j) + u^n(i-1,j) + u^n(i,j+1) + u^n(i,j-1)) - \frac{1}{4}f^n(i,j) \cdot \Delta x^2. \tag{2.11}$$

With the Jacobi method, a perturbation of $u^{n+1}_{i-1,j}$ will be felt on $u_{i,j}$ only after the whole mesh is swept since it will occur for the first time at the next iteration

Algorithm 1 Gauss-Seidel solver pseudo-code.

1: assign_initial_values(); // assign values to the mesh points
2: ... // more initialisation steps
3: // iterations until convergence or T_{max} criteria is satisfied
4: **for** $t \leftarrow 1, T_{max}$ **do**
5: // update all the values of u in the discrete domain
6: **for** $i \leftarrow 2, I_{max} - 1$ **do**
7: **for** $j \leftarrow 2, J_{max} - 1$ **do**
8: $u[i][j] = 0.25 \cdot (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - f_*[i][j])$
9: **end for**
10: **end for**
11: check_convergence()
12: **end for**

through the equation for $u_{i,j}^{n+2}$. With the Gauss-Seidel method, this influence already appears at the current iteration since $u_{i,j}^{n+1}$ is immediately affected by $u_{i-1,j}^{n+1}$ and $u_{i,j-1}^{n+1}$.

From almost every point of view, Gauss-Seidel iteration has proved to be more advantageous than the Jacobi method. For instance, it can be observed that as soon as a new value $u_{i,j}^{n+1}$ is calculated, the 'old' value $u_{i,j}^{n}$ is not needed anymore. Hence, the new value can be stored in the same location and overwrite the local value $u_{i,j}^{n}$. Therefore only one vector u of length n has to be stored, while two vectors u^{n+1} and u^n have to be saved in the Jacobi method. Moreover, the convergence is guaranteed for Jacobi method only if a system matrix is diagonally dominant, while for Gauss-Seidel it needs to be positive definite and symmetric. However, the Jacobi solver is easier to parallelise, as will be demonstrated in Subsection 2.3. The two easy-to-remember schemes used in these methods are visualised in Fig. 2.4.

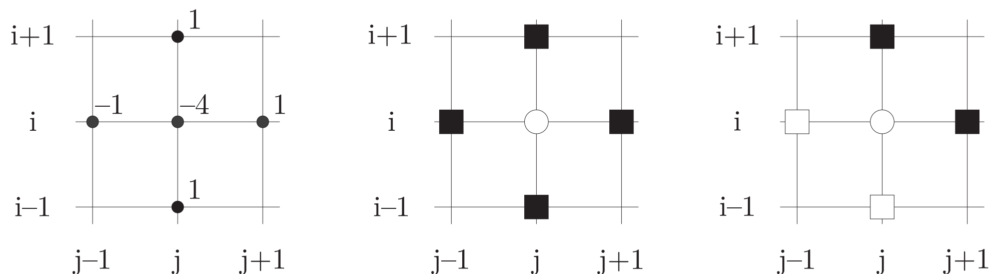


Figure 2.4: left: stencil; middle and right: Jacobi and Gauss-Seidel scheme, respectively – filled nodes represent the values from the nth iteration, while white nodes represent the value from the iteration $n+1$; circle represents the currently updated value

Multigrid methods

Multigrid (MG) methods [50, 82, 51, 163] are an advanced family of numerically stable and computationally efficient methods for iterative solving of linear systems. They are often based on simpler iteration schemes like Gauss-Seidel or Gauss-Jacobi, which is used in the example of a full multigrid below. Like hierarchical (multilevel) preconditioners, multigrid techniques use a hierarchy to accelerate the reduction of low-frequency errors. However, in order to ensure that both high- and low-frequency errors are smoothed quickly, multigrid techniques perform additional smoothing at each hierarchy level. While multigrid techniques can be used as stand-alone iterative solvers (like Jacobi and Gauss-Seidel), it is now common to use certain forms to aid the convergence of an iterative solver, such as Conjugate Gradient. Such methods are particularly effective because they handle the large-scale trends at coarse resolutions.

One full iteration, from fine mesh to coarse and back, is typically called a *V-cycle*. In *V-cycle* algorithms one proceeds from top (finest grid) to bottom (coarsest grid) and back up again (Fig. 2.5, right). On each grid but the coarsest, a relaxation step is done some number of times before transferring (restricting) to the next-coarser grid and also some number of times after interpolating and adding the correction [180].

As a first step in so-called *full multigrid*, the mesh is coarsened and the problem is transfered to the coarsest grid, where it is solved. The solution is then interpolated to the second-coarsest grid and a V-cycle is performed. This is recursively done until the finest grid is reached (Fig. 2.5, left).

If the given discretisation (for the Poisson equation with Dirichlet boundary conditions, on a unit square, e. g.) is based on a uniform grid g with mesh-cell size h, to apply a *V-cycle*, both coarse grids and the intergrid transition operators have to be defined. According to [180], a coarse grid matrix g^* of $(n\times m)$ with a mesh-cell size h can be calculated by full weighting of corresponding $(2n\times 2m)$ fine grid values g by applying local averaging ($i \in [1,n], j \in [1,m]$), e. g.:

$$\begin{aligned}\left(g^h\right)^*_{i,j} = \frac{1}{16}(&g^h_{2i-1,2j-1} + g^h_{2i-1,2j+1} + g^h_{2i+1,2j-1} + g^h_{2i+1,2j+1} \\ &+2(g^h_{2i-1,2j} + g^h_{2i+1,2j} + g^h_{2i,2j-1} + g^h_{2i,2j+1}) + 4g^h_{2i,2j})\end{aligned} \tag{2.12}$$

On the other hand, switching from a coarse grid g of $(n\times m)$ with a mesh-cell size $2h$ to a fine grid g^{**} of $(2n\times 2m)$ by traversing g pointwise for the now “refined” grid indices in x and y direction ($i \in [1,n], j \in [1,m]$), a common choice according

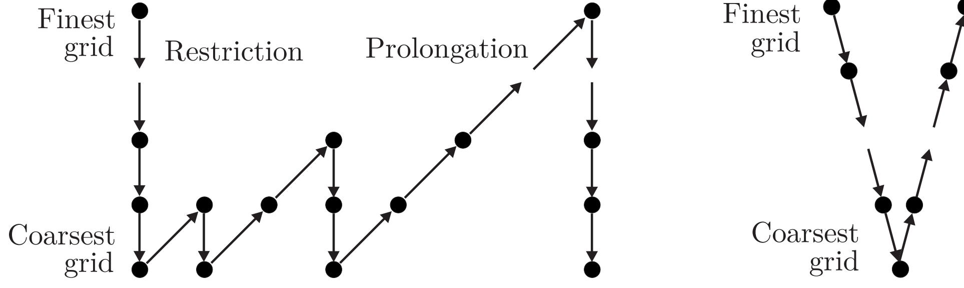


Figure 2.5: Multigrid: left – full multigrid, right – V-cycle

to the same source is bilinear interpolation, as can be seen in Eq. 2.13.

$$\left(g^{2h}\right)^{**}_{i,j} = \begin{cases} g^{2h}_{i/2,j/2}, & \text{if } i \text{ is even and } j \text{ is even} \\ \frac{1}{2}(g^{2h}_{(i+1)/2,j/2} + g^{2h}_{(i-1)/2,j/2}), & \text{if } i \text{ is odd and } j \text{ is even} \\ \frac{1}{2}(g^{2h}_{i/2,(j+1)/2} + g^{2h}_{i/2,(j-1)/2}), & \text{if } i \text{ is even and } j \text{ is odd} \\ \frac{1}{4}(g^{2h}_{(i+1)/2,(j+1)/2} + g^{2h}_{(i-1)/2,(j+1)/2} & \\ +g^{2h}_{(i+1)/2,(j-1)/2} g^{2h}_{(i-1)/2,(j-1)/2}), & \text{if } i \text{ is odd and } j \text{ is odd.} \end{cases} \tag{2.13}$$

Sometimes, for particular applications, it is enough to perform only a part of the *V-cycle*. In [112] only the second half in a pure upsampling procedure is implemented, while in [46] a purely coarse-to-fine solver is used, also called *cascadic*. In [157] it was demonstrated that a cascadic approach has applications well beyond the adjustment of tonal values, such as gradient-based image processing. The Poisson solver propagates sufficient information from coarse to fine, allowing to achieve local solutions at interactive rates that are virtually indistinguishable from the full-resolution solution.

With today's simulations of complex phenomena, no matter how efficient the underlying methods and implementations are, good parallelisation strategies are still a prerequisite rather than an option. In the subsections to follow, different high-performance computing aspects are elucidated.

2.3 High Performance Computing (HPC)

The performance of modern applications increases rapidly due to faster hardware, more available memory, more efficient algorithms, optimisation techniques, and, finally, parallel computing techniques. The main goal of parallel computing is to try to answer the insatiable demand of complex problems for yet more computing power – numerous examples worldwide can be found in climate or geophysics simulations (tsunami, e. g.), structure and flow simulations, large data analysis, military applications, etc. Based on the published booklet results of numerical simulations on the High Performance Computer in Bavaria (HLRB II), an SGI

Altix 4700 system in the period 2009-2011 [171], the HPC projects involved astrophysics, computational chemistry, geo sciences, high energy physics, life sciences, and solid state physics.

2.3.1 State of the Art Parallel Computer Architectures

Having HPC hardware to run simulations on is only a starting point. The rough classification of parallel computers according to the level at which the hardware supports parallelism assumes, on the one hand, multi-core, many-core, simultaneous multithreading (SMT), and multi-processor computers, having multiple processing elements within a single machine (typically, shared memory) and, on the other hand, massively parallel processors (MPPs), and Grids, using multiple machines to work on the same task (distributed memory system). As networks have become faster and multiprocessors more loosely connected, those clear distinctions between the use of shared memory and message passing as communication mechanisms have been obscured. It is worth mentioning the three major types of multiprocessors which have gained commercial acceptance: UMAs (uniform memory access multiprocessors), NUMAs (non-uniform memory access multiprocessors), or their cache coherent counterparts – cc-NUMAs and NORMAs (no remote memory access).

In a shared memory system, all processors in the system share equal access to one or more banks of memory. All areas in memory are equally accessible to all processors and processor-to-processor data transfers are done using shared areas in memory. All processes have direct access to shared variables, thus, in general, they have to be synchronised.

Multicore architectures, as the name indicates, have multiple processing units, so-called *cores*, on the same chip – connected by the on-chip central intersection through which all the information must flow between those cores or between cores and memory and I/O ports. To be considered multi-core, each core has to be a full processor (e. g., a superscalar design, i. e. one core having multiple integer calculating units, together with the enabled pipelining, would not count).

Multiprocessor architectures have more than one processor on a motherboard (each of which in turn can also be multicore).

Manycore architectures have a “revolutionary” relatively new chip design, which can harness the processing power of tens to hundreds [7] or even thousands [32] of cores on a single chip interconnected with on-chip network. It is considered a processor in which the number of cores is large enough that traditional multiprocessor techniques would no longer be efficient. In other words, a manycore architecture is characterised by more simple, low-power cores coupled with high memory bandwidth.

SMTs have an even smaller subset of a core’s components duplicated. An SMT core has duplicate thread scheduling resources, so that the core looks like two

separate “processors” to the operating system. In fact, it has only one set of execution units. One such implementation is Intel’s Hyperthreading.

Uniform memory access multiprocessors (UMAs or cc-UMAs, using special-purpose hardware to maintain cache coherence) are those with access time to a memory location nearly independent of which processor makes the request or which memory chip contains the data.

For *non-uniform memory access multiprocessors* (NUMAs or cc-NUMAs), the memory access time depends on the memory location relative to a processor – a processor can access its own local memory faster than non-local memory (i. e., memory local to another processor or memory shared among the processors).

No remote memory access (NORMAs) are those with message-based communication among processors over an internal or external network.

In a pure distributed memory system, each processor in the system has its own separate bank or banks of memory. Processor-to-processor data transfers are done over some form of network topology.

MPPs use of a large number of processors (or separate computers) to perform computations in parallel.

Grid computing is done on a large number of heterogeneous computers in geographically distributed and diverse administrative domains. The resources are opportunistically used whenever they are available.

Some facts about HPC

“The trend of increasing a processor’s speed to get a boost in performance is a way of the past. Multicore processors are the new direction manufacturers are focusing on. (...) Throughout the 1990’s and the earlier part of this decade microprocessor frequency was synonymous with performance (...) Since processor frequency has reached a plateau, we must now consider other aspects of the overall performance of a system: power consumption, temperature dissipation, frequency, and number of cores. Multicore processors are often run at slower frequencies, but have much better performance than a single core processor.” [149]

Namely, a co-founder of Intel, Gordon E. Moore, stated in 1965. that the number of transistors that could be placed on an integrated circuit was increasing exponentially, doubling approximately every two years (or eighteen months, according to Intel executive David Hause). The trends in HPC architectures are directed also by the fact that 25% reduction in performance (i. e. core voltage) leads to approximately 50% reduction in dissipation. Therefore, manufacturers came up with the idea of installing two cores per die, having together the same dissipation as single core system, but better overall performance. However, if writing a program which can be run only on one core on such architectures, this non-parallel program will get slower. In fact, even when a program *is* executed on multiple cores, the increased number of transistors and the number of cores per die unfortunately still does not give equivalent performance gains.

One of the concerns is memory/cache coherence. Distributed and shared caches on the chip must make sure that when one core reads data from memory, it is reading the current value, not one that has been updated only in the cache of a different core. All the cores need to communicate with each other, and to the main memory. This is usually accomplished either using a single communication bus, or an interconnection network, so the answer of manufacturers is to increase the bandwidth, with on-chip memory controllers and interconnects. With a shared cache, if there is no proper replacement policy, one core may starve for cache usage, thus, continually make calls out to the main memory.

Regarding memory itself, rather than increasing its capacity, there is an urge for re-design of the interconnection network between the cores. That is, decreasing intercore communication latencies should not be underestimated.

Finally, using aforementioned processors and clusters to their full potential is another issue. If the code developers do not write their programs so that their parts can be executed concurrently on multiple cores, and especially without heavy synchronisation among those parts, there is no gain in performance. "The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years." Intel fellow Shekhar Borkar said in 2007. [85]. The question is how many code developers in today's scientific and engineering applications understand the necessity to strive towards that goal and how capable they are of achieving it.

2.3.2 Parallel Programs

To "support parallelism" means that a program itself must be written in such a way that the execution of program instances occurs on more than one core/CPU/node. This often requires major changes or rewriting of the corresponding sequential algorithms, yet is the only way to take advantage of parallel computing architectures, i. e. increase the computational performance relative to the performance obtained by the sequential execution of the program on one core. Keeping this in mind, the best parallelisation strategy and implementation for a specific program should be chosen, taking into consideration not only the problem itself (its algorithms, adaptability of methods and data structures to the problem; the ways in which one can decompose it into concurrent parts), but also the available computer resources. Having a program portable to other machines without a significant decrease in performance is certainly itself an important issue.

The typical necessary steps to be taken are: identify concurrent tasks, map tasks to different processors; manage inputs, outputs, and other data; manage shared resources; synchronise the processors, and so forth. Further steps, addressing concurrency in particular – potential resource starvation states which should be avoided (deadlocks, livelocks, etc.) or synchronisation steps – might need to be applied. They are commonly implemented via synchronisation primitives, such as semaphores, mutexes (short for mutual exclusion) or barriers.

The main objectives are one or all of increasing throughput, shortening the response time, or increasing the possible problem size. As the simplest example, known as *embarrassing parallelism*, N instances of a sequential program can be simultaneously run with different input data. More common approaches would be: (1) computing one problem by running N instances of a parallel program for jointly solving different problem tasks, or (2) computing one problem by running N instances of a parallel program and N-times larger data, using the sum of all local memories for computing.

Depending on the goals of the software developer, there are two major approaches for breaking the problem into concurrent tasks, i. e. decomposition. These are *data* and *task* approaches for decomposition, although this distinction is not necessarily a strict one.

2.3.3 Decomposition

There are actually many ways to decompose the problem in order to utilise parallelisation for a specific problem. Roughly speaking, there are two most common kinds of decomposition: domain decomposition (data parallelism) and functional decomposition (task parallelism).

There is a Russian saying which rougly translates to: "Do not forget that you can eat an elephant if you slice it into small enough portions." In *domain decomposition*, data is divided into pieces that are approximately the same size and then mapped to different processors. Each processor works only on the portion of the data that is assigned to it. However, the processes may need to communicate periodically in order to exchange data (e. g. on the borders of sub-domains).

For some large, complex problems, however, domain decomposition may not be the most efficient strategy for a computation. This is the case, for instance, when the individual subsets of data assigned to the different processes require greatly different amounts of time to be processed. The performance of the code is then limited by the speed of the slowest process since the remaining processes are idle. In these situations, *functional decomposition* can be considerably more efficient. Using this strategy, the problem is decomposed into a large number of smaller functions, which are then forwarded to the processors as they become available so that processors that finish quickly are simply assigned additional work.

Besides this rough classification, there are a few already established decomposition methods, worth being named at this point:

- Recursive task decomposition – one problem is split into tasks which can be done completely concurrently, each processor getting its own part. In the next recursion step, parts are again split among processors. The disadvantage may be that with a large number of recursion steps, the number of processes which can be utilised may decrease, i. e. on the lower levels of recursion.

– Data decomposition – typically used when large data structures need to be processed. For instance, matrix data can be split for concurrent computation row-wise, column-wise, or block-wise (Fig. 2.6 left, middle and right, respectively).

– "Hybrid" decomposition has also become popular. Namely, first a recursive decomposition is done, so that then the tasks' data and computations can be split where needed. This is well illustrated in the *Bone* application test scenario, at the point where distributed parallel scenario and its improvements are discussed, and also sketched in Fig. 2.7.

– Exploratory decomposition is employed in search algorithms.

– Speculative decomposition – for dependent choices in computations; but the last two named are neither employed as part of this work, nor in the application scenarios, thus, are just mentioned for the reason of completeness.

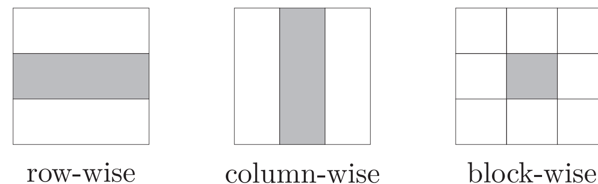


Figure 2.6: Data decomposition – left: row-wise, middle: column-wise, right: block-wise decomposition of a matrix. Each filled part represents an example of the domain which is assigned to a single processor.

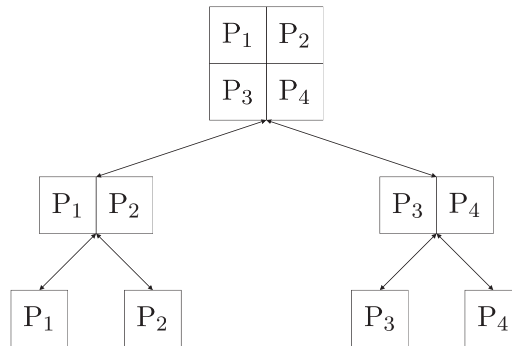


Figure 2.7: Hybrid decomposition – recursive decomposition of the tasks where each task in the hierarchy is split based on data decomposition. The tasks are assigned to different processors P_i, $i \in \{1, \dots 4\}$.

Often computational effort required for parallel simulations does not allow for results to be gained quickly enough for (especially high-rate) user interaction. When custom decomposition techniques are not enough to fully exploit the hardware available, sophisticated data structures and methods are required [56, 130].

Hierarchy based structures, such as trees, are very convenient due to their spatial subdividing property [56, 130]. For instance, octrees are based on the principle of successively dividing a cube where needed in every direction. The resulting cells either lay completely inside or outside that geometry, at least due to some threshold value, or are cut by the boundary. Fig. 2.8 illustrates the 2D variant of this, creating a quadtree. Examples of both 2D- and 3D-spacetrees can be found in [56], and collections of adaptive trees (conveniently called forests of octrees) in [57]. Making use of this structure to store the volume information (each cell stores some), the complexity can be reduced to O(N^2) cells in 3D or O(N) cells in 2D instead of O(N^3) or O(N^2), respectively, in case of an equidistant discretisation.

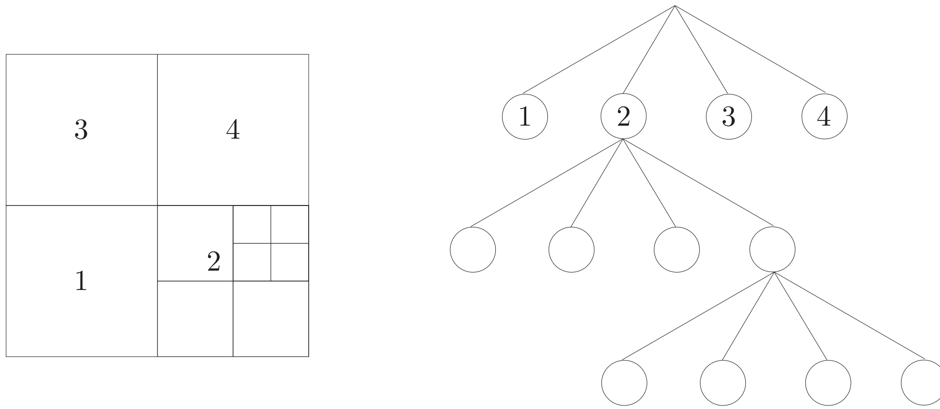


Figure 2.8: Nested dissection of the geometry domain: (left) halving the domain in each spatial direction. The cells either lay "inside" or "outside" of the specified geometry (at least due to some threshold value), or they are cut by the boundary; (right) the corresponding quadtree structure.

Alan George was the first to apply nested dissection (ND) on finite element problems [78], in the 1970's. As described in [130], his idea was to subdivide a whole domain Ω recursively into sub-domains Ω_i and after eliminating all the unknowns which are local for Ω_i to solve the remaining, so-called Schur complement, system of equations. Now that the sub-systems are released from mentioned influences of local variables they may be processed independently from each other (Fig. 2.9).

Let us consider the system in Eq. 2.5. Approximately halving the system matrix in each dimension creates four sub-matrices K_i, $i = 1, \dots 4$ as an outcome. If we arrange these matrices so that they can be divided into inner parts (II), containing only local variables, outer parts (OO) and inner-outer parts (IO, OI), the systems now look like what is shown in Fig. 2.10 (A, x and b are arranged counterparts to K, u and d respectively, $i \in 1, 2, 3, 4$).

After substitution of x_I from the corresponding Eq. 2.14

$$\begin{aligned} A_{II} \cdot x_I + A_{IO} \cdot x_O &= b_I, \\ A_{OI} \cdot x_I + A_{OO} \cdot x_O &= b_O \end{aligned} \tag{2.14}$$

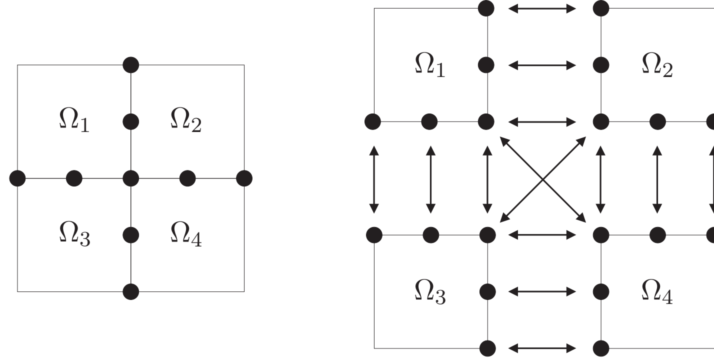


Figure 2.9: Domain decomposition – domain Ω divided recursively into sub-domains Ω_i, which can be processed independently except on the common boundaries (represented with the filled circles).

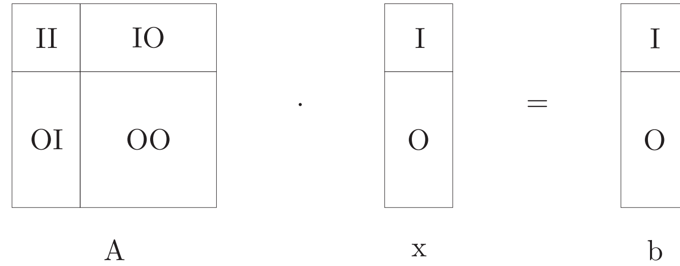


Figure 2.10: Division of the problem into II – inner-inner, IO – inner-outer, OI – outer-inner and OO – outer-outer parts depending if the corresponding degrees of freedom are local or global (shared among different elements). A is a matrix of $(n \times n)$, and x and b are vectors of the size n.

and doing few obvious transformations, it looks as follows

$$(A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}) \cdot x_O = b_O - A_{OI} \cdot A_{II}^{-1} \cdot b_I, \tag{2.15}$$

where $A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}$ is called the *Schur complement*.

Denoting $A' = A_{OO} - A_{OI} \cdot A_{II}^{-1} \cdot A_{IO}$ and $b' = b_O - A_{OI} \cdot A_{II}^{-1} \cdot b_I$, the following system is gained:

$$A' \cdot x_O = b'. \tag{2.16}$$

In Eq. 2.16 the influences of local variables are eliminated. The same principle may be used for each of the sub-matrices gained in the potential further recursive subdividing of the matrix K. Finally, all the Schur complements of "small" matrices and right-hand sides are assembled, resulting in a new system of equations, whose matrix is the assembled Schur complement. It can be solved, for example, using a direct solver, such as Gaussian elimination. In this way the vector x_O of unknowns is calculated and a proper substitution of unknowns with its values is done in the first equation in Eq. 2.14. From Eq. 2.14 the rest of the

solution – x_I – is calculated

$$x_I = A_{II} - 1 \cdot b_I - A_{II} - 1 \cdot A_{IO} \cdot x_O. \quad (2.17)$$

Despite the fact that, as mentioned before, the efficiency of this solver compared to other direct solver options is better, employing an iterative method with preconditioner, or even one of the multigrid methods should be considered. The ND solver is, nevertheless, employed in one of the framework application scenarios related to a femur simulation. It is a convenient example to demonstrate domain decomposition techniques, where Schur-complement-based methods are the standard and typically allow for efficient parallelisation. However, the challenge of implementing a scalable parallelisation strategy becomes greater in the case of "long" structures, as in the femur example, due to the resulting unbalanced octree data structures, and the corresponding tasks with dependencies.

Dependency issues

Once the concurrent tasks are identified and the problem is decomposed as desired the need for synchronisation and data access issues must be considered.

First, a more obvious category, are task dependencies (especially for task decomposition of the problem). These impose constraints concerning the order of execution of tasks: whether or not they fulfill the conditions related to other tasks.

Second, a data-decomposition-based parallel program might also have execution-order constraints between statements (i. e. instructions) due to dependencies. Hence, dependency analysis should determine whether or not it is safe to reorder or parallelise these statements.

A *task interaction graph* is formed by considering nodes to be tasks and edges their interaction. One can distinguish between data dependencies (including loop dependencies), and task dependencies, together with control dependencies.

A *task dependency graph* is a sub-graph of the task interaction graph. A task decomposition of the problem can be illustrated as a directed graph, where again nodes are tasks, and edges dependencies among them. A dependency between tasks implies an interaction between them.

In a sense similar to task dependencies, *control dependencies* are defined as follows: an instruction (or a block of code) executes if the previous instruction (block) evaluates in a way that allows its execution. This is essential in dataflow models, which will be described in Subsection 2.6.2.

On the other hand, *data dependencies* arise due to competitive access to shared data, which often entails synchronisation. Data dependencies might lead to inefficiencies and bottlenecks, hence preventing optimisations such as out-of-order execution or parallelisation. Modern tools, such as KAP pre-processors, developed by Kuck & Associates, Inc. (KAI) use dependence graphs to find potential dependency issues and examine them to see if those ties can be "broken".

These dependencies occur for two tasks when: (1) task T_1 modifies a resource that the other task T_2 reads and T_1 precedes T_2 in execution, (2) if T_1 and T_2 modify the same resource and T_1 precedes T_2 in execution [41]. Case (2) may be removed through renaming of variables.

Since all the applications which served as a basis for integrating the interactive computing framework use some kind of parallelism (often several combined), all the named concurrency and dependency aspects have been carefully considered in their implementations – especially while integrating the interactive computing framework (where the task, as shown later, becomes more challenging).

2.3.4 Mapping of the Tasks

The most relevant, sometimes conflicting, aims of a good mapping of tasks to processors is to maximise concurrency by mapping independent tasks to different processors, minimise idling by even load distribution and minimise interaction, i. e. communication, overheads among tasks by mapping interacting ones on the same process. Thus, the assignment's goal is to find a good trade-off strategy.

Static mapping

For this kind of mapping, the decisions about the distribution of the tasks is done prior to program execution time. Jobs are defined at compile/link time, based typically on problem size, number and performance of processors, amount of data to communicate, etc. This technique is used mostly for the *a priori* known hardware systems with known characteristics and connection topologies. Yet, an application which might be perfectly balanced based on analysis, can turn out to be much different in practice. Past performance results can be used to improve future performance. However, processor loading due to other users/programs, or inherent communication and synchronisation delays of the system cannot be taken into account, nor can a shutdown of one of the resources be resolved by migrating mechanisms.

Most common static mapping strategies are:

– *Based on data partitioning*
– *Based on task partitioning* – one example are hierarchical structures such as an octree of tasks. It is crucial to minimise the interaction overhead by mapping interdependent tasks onto the same processors (i. e. tasks along the "straight" branch). Inevitable idling in bottom-up processing has to be compromised somehow.
– *Hierarchical mappings* assume a combination of the two. In the case of different weights for the tasks, each big task is considered a "supertask", for which classical data decomposition can be done.

Static mappings might be preferred when there is large data compared to computation, thus, e. g. migration of the tasks is costly. Even static mappings may be

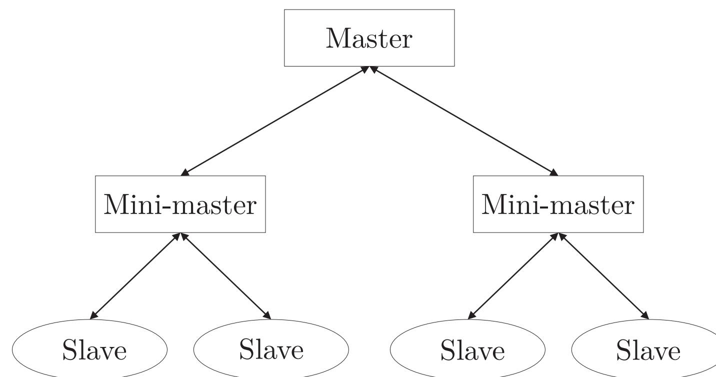


Figure 2.11: Hierarchical master-slave structure: the master process splits its responsibilities with several "mini"-masters, each of which becomes in charge of a certain number of slaves.

challenging – the problem of obtaining an optimal mapping is an NP-complete problem [109] for non-uniform tasks. In practice, simple heuristics provide good mappings, as will be shown in Chapter 4.

Dynamic mapping

For this kind of mapping, tasks are generated dynamically at program execution time since the task sizes are not known *a priori*. Jobs are assigned to corresponding processors, but the work load is adjusted during run time. The cost of moving data may outweigh the advantages of dynamic mapping. In a shared address space dynamic mapping may work well even with large data, however careful consideration of the underlying architecture is necessary (e. g., NUMA or UMA) because it might occur that the data needs to be moved physically in this case as well. It is a more commonly used type of mapping.

Among dynamic mappings, one typically distinguishes between:

- *Centralised schemes* (master–slave, self–scheduling, chunk scheduling) – where there is a central entity where jobs are handed out or tasks are placed into a single work pool. For the master–slave concept, it is possible to employ several master-levels – "mini-masters" to try to avoid bottlenecks (Fig. 2.11), but there is a reasonable fear of then additional synchronisation, communication and computation complexity that this implies. Another question is which data is worth keeping in the memory for which process and which data is always communicated among them.

- *Distributed schemes* – the load balancer is located on all processors. The performance information has to be broadcasted throughout the system. Central coordination is not needed since all the processors have all the information they need.

What may be concluded from the previous content is that while decomposition is done based on the actual concurrency among tasks within a specific problem, the overall performance depends heavily on the mapping strategy.

2.3.5 Load Balancing (LB)

To keep the workload distributed evenly among available resources involves a trade-off between balancing the workload and the price one pays for communication. The load balancing is defined by the decomposition of the initial problem, and the way the resulting components are mapped to different processors.

Thus, additional runtime effort is required to support running the balancing algorithm. Typically those algorithms have to communicate the information about the current load among the processors, recalculate the load and redistribute jobs when needed.

The best load balancing strategy is dependent on the type of application. To find an optimal strategy, it is reasonable to consider: (1) if the system is based on an analogy to physical systems (physical, probabilistic models, etc.); (2) the topology of the target architecture (bus-based shared memory multiprocessors, workstation clusters interconnected by networks, distributed memory multiprocessors, whether or not they are of a homogeneous or heterogeneous architecture); (3) the network topology (grid-like, tree-like, non-interacting), to have components connected in an appropriate way; (4) whether nodes which communicate with each other are direct neighbours, or they are close to each other; (5) if the transfer of load-state information is system-wide or restricted, what is the extent to which processes (load balancers) collect load info, before making a decision: partial or complete; (6) if the decision structure is centralised, distributed or hierarchical, decision mode – autonomous, cooperative, competitive; (7) if initiation of communication is done by the sender (e. g. when it is overloaded), or receiver (when a target is underloaded); (8) if a central LB entity exists, or it is timer-based or threshold based; (9) migration costs; and many more.

2.3.6 Parallelisation and Communication APIs, Libraries and Middleware

Once the load balancing strategy is chosen, the question which arises is – how a program should be written, compiled and executed, such that multiple program instances happen concurrently on multiple cores?

When a memory-coupling parallel model on shared memory systems is used, the programs written in a sequential language are mapped to the parallel computer via compiler directives, library calls, etc. They are used, e. g. for splitting the scope of loop iterator indexes and the corresponding tasks among different computational resources. One example of thread-based parallelism (*multithreading*) is that a single processor executes code between loops, but activates a set of processes to cooperate in executing a parallel loop, called a *fork*. A thread then may

be *active* (running), *sleeping* (pausing) for a specified interval, or *yielding* (being placed at the end of the run priority queue and another thread is scheduled to run). Threads can be synchronised by explicit or implicit *barriers*, where they *join* each other, or gain exclusive access to some part of the code – a *single* thread, or one thread at a time via:

- *scoped locking*, i. e. *mutexes* and *locks* of different libraries (where it is allowed to leave the locked region with *break*, *continue*, *return*, or
- *critical sections* via an OpenMP threading interface, e. g. (which are not exception safe and *break*, *continue*, *return* are forbidden within – which is a point of interest for the framework implementation, as shown in Chapter 3).

Most of the aforementioned functionality is supported by OpenMP directives, POSIX threads (pthreads) library calls, the Boost threads library, etc.

OpenMP (Open Multiprocessing) is one of the most commonly used application programming interfaces (APIs) for writing multithreaded programs, using a set of compiler directives, (runtime) library routines, environment variables, etc. It is available for C, C++, and Fortran suited for programming UMA and SMP systems, DSM/VSM systems (i. e. NUMA, cc-NUMA), and hybrid multiprocessor systems in combination with message passing (e. g., Message Passing Interface – MPI).

In distributed memory systems, a message-coupling model is used – the one that extends parallel languages (C/C++, Fortran) by additional parallel language constructs, implemented via procedure/function calls from communication APIs (for instance, MPI).

MPI is a de facto standard communication API providing virtual topology, synchronisation, and communication functionality between a set of processes. Its goals are high performance, scalability, and portability. It belongs to layers 5 and higher of the Open System Interconnection (OSI) Reference Model, with sockets and Transmission Control Protocol (TCP) [8] used in the transport layer by default.

The *point-to-point* MPI communication routines can be:

- *Blocking*: they do not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. The representatives are: *MPI_Ssend* (“synchronous send”), *MPI_Send* (“standard”), *MPI_Bsend* (“buffered”), *MPI_Rsend* (“ready”), *MPI_Recv* (“standard receive”).
- *Non-blocking* calls return immediately, no matter if the data is safely stored in the buffer, allowing to overlap other computation while testing for the

operation to be completed – examples are *MPI_Isend*, *MPI_Ibsend*, and *MPI_Irecv*.

In addition to this, send/receive operations can be classified as either *synchronous* or *asynchronous*, depending on if they can return before the matching receive/send have reached a particular phase.

Apart from aforementioned point-to-point rendezvous-type forms of send and receive operations (the rendezvous protocol is elaborated in Chapter 4), MPI supports: choosing between process topologies; collective communication, such as broadcasting a message to a group of processors (*MPI_Bcast*), combining partial results of computations (gather and reduce operations); synchronising processes (barrier operations) as well as obtaining the number of processes in the session; current processor identity that a process is mapped to; neighbouring processes accessible in a logical topology; and so on.

Although MPI programs are portable to different platforms, both shared and distributed memory supercomputers, different MPI implementations are typically optimised for the hardware on which it is used. In MPI-3 standard, attention is paid to the fact that there will be more multicore and hybrid architectures, thus, non-blocking collective communication is suggested to be implemented. For instance *Ibcast* (non-blocking broadcast) is planned. Early discussions of the MPI Forum lead to the conclusion that nonblocking collective operations introduced too many complexities and instead recommended the use of separate threads that make blocking calls. Due to the ever-growing number of machines within clusters in computer centres, fault tolerance will also be considered. It is still unknown which functionality announced at this point will be part of the new standard, will be implemented in all MPI distributions, and will have consistent performance.

On distributed shared memory based architectures it is advantageous to mix message passing and multithreading to maximise performance [101]. This approach is referred to as "hybrid" or "multilevel parallel programming".

When using multithreading and MPI calls, the level of support for send and receive calls in different threads has to be considered:

- MPI_THREAD_SINGLE – only one (user) thread is supported,
- MPI_THREAD_FUNNELED – many user threads, but only one thread may make MPI calls,
- MPI_THREAD_SERIALIZED – many user threads may make MPI calls, but only one thread at the time does it (user has to guarantee this),
- MPI_THREAD_MULTIPLE – free for all, any thread may make MPI calls at any time.

An MPI implementation of MPI_Init_thread is allowed to return any of these values (which indicates the level of support provided on a system). For example, an MPI implementation that is not thread safe will always return

MPI_THREAD_SINGLE. The user may, however, choose whether to pay any performance penalty that might come with a fully multithreaded MPI implementation.

What is relevant for this work, as it may be seen later, is that threads provide a natural implementation of nonblocking communication operations [80]. A thread can be created to do a blocking receive operation, e. g. As long as this blocks only one thread and not the whole process, it has the effect of a nonblocking receive. The same applies to sends. In fact, it is often preferable to use threads and the blocking versions of the sockets on those Unix systems that support threads, as this provides a simpler programming model. Additionally, threads can increase the convenience and efficiency of the implementation of collective operations.

In order for threads to be used in combination with a library, however, the library must be designed to work well with threads. This property is called *thread safety*. Thread safety means, in the case of a message passing library, that multiple threads can be executing communication, synchronisation, etc. calls without interfering with one another. Thread hazards occur when a message-passing system is expected to hold certain parts of the process state and it is impossible to hold that process state for more than one thread at a time. For example, some libraries use the concept of "the most recently received message" to avoid passing a status argument stored on the process's stack.

For an application to use MPI with threads, it is not enough that the implementation of MPI is thread-safe. As [80] explains, the thread library must be aware of the MPI implementation to the extent that execution of a blocking operation will cause the current thread to yield control to another thread in the process, rather than cause the process to block. When a message arrives, a thread waiting for it should be made runnable again. Furthermore, when a system call is made, the operating system should block only the thread that made the call, not all threads, or much of the flexibility of user-level threads is lost [80]. This is very important for the implementation of non-blocking communication pattern using blocking MPI calls in multiple threads, as in some of the application test cases.

2.3.7 Examples

Let us consider a simple example from Section 2.2, now from the parallel computing perspective. To parallelise Jacobi algorithm for the Poisson equation discussed before, there is a choice between shared-memory and message passing based methods. The basics of parallel programming as well as the solver techniques are already provided in this chapter. The framework is tested on both of the methods in terms of the induced overhead, as shown in Chapter 3. Therefore, it is helpful to illustrate both of the parallelisation methods for this example.

Both of them can be done column-wise or row-wise (illustrated in Fig. 2.6). An implementation of a multithreaded program (using OpenMP directives) to solve the Poisson equation by Jacobi iterative method is very straightforward, as shown in Algorithm 2.

Algorithm 2 Jacobi pseudo-code – multithreaded version.

1: assign_init_values()
2: // iterations over time
3: **for** $t \leftarrow 1, T_{max}$ **do**
4: // here start parallel(i) region, private vars(j)
5: **for** $i \leftarrow I_{min}, I_{max}$ // each thread has its own $[min, max]$ **do**
6: **for** $j \leftarrow 1, J_{max}$ **do**
7: $u[i][j] = 0.25 \cdot (u_old[i+1][j] + u_old[i-1][j] + u_old[i][j+1] + u_old[i][j-1] - f_*[i][j]$
8: **end for**
9: **end for**
10: check_convergence()
11: copy_u_to_u_old()
12: **end for**

If the program is written for a distributed memory machine using MPI, many questions arise. Which tasks are concurrent? How should we distribute the data (the data locality issue), or rather – how should we distribute a coefficient matrix and a right-hand side among CPUs? How should we distribute a vector of unknowns? What data needs to be communicated?

In the case of row-wise distribution, assuming without losing generality that a dimension of the matrix can be divided by the number of CPUs, the blocks of m rows of the matrix u_old – the system matrix from the previous iteration – would be distributed to different CPUs. The vector f – the right hand side of the Poisson equation – would be distributed similarly.

Since at the end of one iteration only part of the rows of u_old are available on each CPU for the next one, the "ghost" layers (the neighbour rows "belonging" to other CPUs) need to be communicated after each iteration. It is simple to implement, but involves much communication and therefore limits (especially strong) scalability. Using techniques for overlapping computation and communication could improve scalability.

Namely, each CPU needs to exchange a vector with the neighbour above and below. In order to overlap communication with computation (see Fig.), each CPU does the following: first posts non-blocking requests to send and receive data to/from upper neighbour and to receive/send data from lower neighbour. These return immediately, which allows the next step: partial computation with the data currently available. Finally, the process checks non-blocking communication status and waits if necessary, before repeating the whole sequence.

In the case of column-wise distribution, blocks of m columns of a coefficient matrix A are distributed to different CPUs; each CPU has to keep the whole f vector. Since the data has to be communicated among processes, with vertical ghost-layers this time, it can be overlapped with computation, similarly to the previously described scheme.

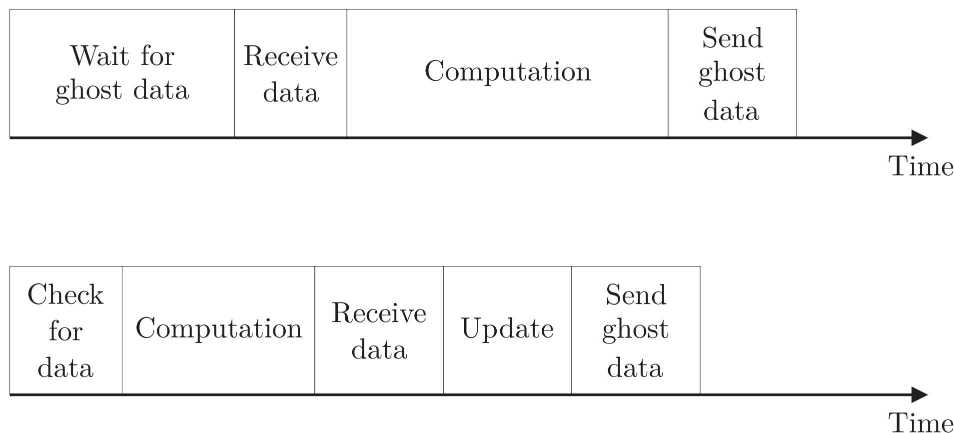


Figure 2.12: Above: one iteration consists of waiting for the necessary data from the ghost neighbour entries, computation, and sending the processes' own ghost data. Below: overlapping the communication, i. e. waiting for the necessary ghost data to arrive, with the computation which does not require that particular data. At the end of the iteration, all the remaining ghost data is received and the remaining entries updated.

An implementation of the block-wise distribution for the Laplace equation has served as one of the test cases for the framework. It is illustrated in Chapter 3 in that context.

2.4 Postprocessing and Communication

When scientists and supercomputers are in geographically different locations, the most significant barrier to effective visualisation, especially within interactive computing, is that networks cannot cope with the amount of data to be transmitted. In shared memory systems, the main bottleneck is often related to memory access. In distributed systems, sometimes many instances of the same program execute on different, possibly heterogeneous computing platforms and exchange data when needed. Or even several *different* components are interconnected via network and exchange data over that network – for instance a simulation, a visualisation and a databases component. Each of them may itself perform some form of concurrency. The performance of such a system depends on the execution time of all the executing components, the amount of data which has to be exchanged, the speed of transmission, and also the way the components are synchronised with one another.

Improving the network architecture is critical, however, research must offer techniques to reduce the demand on the network [33]. Thus, it might be beneficial to reduce the amount of transferred data when possible.

Once a satisfactory trade-off between load balance and communication costs is found, a variety of communication strategies should be applied in attempt to

increase the transfer rate. Here, one distinguishes between hardware related strategies, network protocols, or software based strategies, such as postprocessing techniques – selection, compression, re-ordering of the data, etc. Storing data on media attached to the supercomputer is often inadequate due to the amount of data. How to postprocess simulation data on a supercomputer efficiently is a challenge gaining impact due to the rapid increase of the number of cores on supercomputers and, at the same time, slow hard disk I/O when copying data. At least a few trends are studied extensively, hence, are listed in the paragraphs to follow.

Compression: One of the approaches described in [89] is running the simulation and the client in a distributed manner. For easy data exchange, a streaming class has been defined which is used as a container for data of all kinds and which can be compressed during the compression phase.

Selection: Many applications write the whole simulation data to a stream. The data is then interpreted on the client side. This classical approach is often inadequate due to network bandwidth restrictions. Often, most of the data is not required in full resolution, and so multiresolution approaches are fruitfully applied.

Mutiresolution streaming is used in [156]. As soon as the amount of data returned by the query requests exceeds available computing resources, the query results can be returned at different levels of resolution. The indices list for the hierarchy of resolutions can be encoded. When the query is performed, the application can specify the desired level of resolution to be returned as a result.

In [33], data is read from the disk, using a multiresolution respresentation of the data to reduce the amount of data sent over the network. The visualisation starts at a coarse resolution for a quick overview. If a user zooms in to a particular part, the data is refined only for this part, and streamed to the user for visualisation. The rest of the data is left at the coarse resolution. To achieve fast interaction, the rendering process normally takes place on the client's side.

In [37] streaming the dynamically adaptive grids of a particular CFD solver was found to be inadequate, as the visualisation and postprocessing algorithms were written for regular Cartesian grids. A two-fold strategy that overcomes these drawbacks is proposed. The CFD code delivers only data really required for post-processing steps (on-demand) and bandwidth is invested where it does the most good [37]. The post-processing codes pass queries which comprise the spatial region, the resolution and the variable of interest to the simulation. The solver then returns the Cartesian grid of the requested data, which is a low-overhead data-structure, according to [37].

What is often used is a **sliding window model**, such as in [39]. Here, the data elements arrive in a stream and only the last N (i. e. window size) elements to have arrived are considered relevant at the moment. The rest are deemed *expired* and they no longer contribute to query answers or statistics on the datasets [39].

Selection – subsampling and tiling: The system described in [157] also used efficient multi-resolution range queries. The framework only needs to access and solve visible pixels and provides adaptive multiresolution access to out-of-core images in a cache-coherent manner. Moreover, it is flexible enough to handle different hierarchical image formats – both tiling for higher-quality images, and subsampled hierarchies. All these approaches have their pros and cons. The tiled image is visually pleasing, but details are lost. The price to pay is significant preprocessing, reduced flexibility when dealing with missing data, and increased I/O when traversing the data [157]. On the other hand, the least costly image hierarchy can be computed by subsampling. Subsampled hierarchies have aliasing artifacts, but also retain enough contrast to see details [157], in many cases. For greater I/O speed, the HZ-order proposed in [137] is used to stream the data.

Re-ordering: With a minor variation to the underlying I/O layer, the previously mentioned system also supports a faster, subsampled *Hierarchical Z-order* as proposed by Pascucci and Frank [137]. To achieve the level of scalability necessary in the current system, they further simplify the HZ data access scheme [157]. They use a lightweight recursive algorithm that avoids repeated index computations, provides progressive and adaptive access, guarantees cache coherency, and minimizes the number of I/O operations without using any explicit caching mechanisms [157].

Postprocessing: Another approach is in-situ post-processing, where the postprocessing is done directly on the computing node [164]. A simulation runs on a supercomputer, where images are produced from the results during run time and sent to the remote desktop/laptop user client. Such an approach is suitable particularly in heterogeneous environments with graphics cards physically attached to the computing node as postprocessing devices. It also keeps the required bandwidth quite low, by streaming only the data actually studied to the user.

Hardware related techniques: Current leadership-class supercomputers suffer from a significant imbalance between their computational power and their I/O bandwidth [33]. One of the ideas is to ship I/O calls from compute nodes to dedicated I/O nodes [33].

IO forwarding is an approach where the slow I/O operations are deployed to specialised cores to overlap I/O operations. It is described in [170] where the target architecture, IBM Blue Gene/P supercomputer runs a light-weight Linux-like OS on the compute nodes which mitigates OS interference by disabling support for multiprocessing and POSIX I/O system calls. To enable applications to perform I/O, this minimalistic OS forwards I/O operations to the dedicated I/O node, which executes I/O on behalf of the compute nodes [170].

High-throughput communication is used in the *CAVERN* project [44]. They have implemented three mechanisms to increase the throughput of data transfers over a network. For reliable networks such as those using the TCP protocol, when *using multiple connections*, delays caused by waiting for acknowledgement packets can

be hidden by serving other connections that are ready. Another technique refers to *data volume reduction* and *data encoding* (where some of the accuracy is lost on the way – it relies on the fact that the receiving side might not always be interested in the most accurate representation of the data) and *data compression* (via freely available compression libraries).

2.4.1 Network Hardware

To sum up, the communication cost on the simulation back-end, as mentioned before, has to be kept as low, so as not to surpass the benefits of doing the actual computations in parallel. In parallel programs, different network strategies (static/dynamic) can handle communication in between nodes. If hardware is known *a priori*, it is easier to assign the tasks to processors statically. Communicating large simulation results to the user component should be done using some of the aforementioned techniques.

Concerning hardware, connections can be made by standard network cards (100 MBit/1 GBit), or special components can be used: MyriNet 10 Gbit/s, Infiniband up to several hundreds Gbit/s, etc. Inter-node network topology can have major influence on the performance of a parallel program, i. e. whether two nodes which are physically connected with each other are exactly those which need to communicate data among each other (an optimal case). On many of the systems the default mapping of processes to different processors is done, e. g. in round-robin manner, however, there is a way to control this via *processor affinity*. Using this concept, one can also take advantage of the fact that some data may remain in one processor's cache from the previous run, resulting in a reduced number of cache misses.

2.4.2 Challenges

Even if it is given that all the software and hardware components involved in the interactive process are optimal/optimally used, still some challenges and contradictions have remained. Namely, HPC resources are mainly configured to run batch jobs, however, they now need to be utilised in an interactive manner for many large-scale problems.

In [53], for example, an issue affecting the operation of remote steering due to protection by firewalls is solved by system administrators by opening some number of ports to be used for a particular application. According to the same source, the main problem becomes controlling the user environment on (possibly many) heterogeneous machines. The ability provided to "dynamically configure the software to use one of the open ports greatly simplified this process" [53]. They also use a container environment which can be configured and represents a single point of contact for different components.

Another possibility to combine interactive computing and HPC resources is to run 'small' simulations, i. e. with reduced accuracy, on small, interactive cluster

systems for a quantitative analysis, before any promising setup figured out by the user is then (automatically) launched (as a batch job with high accuracy for a quantitative analysis on a supercomputing system) as proposed in [36, 76].

Therefore, yet another conclusion of this chapter is that for a fully operational, portable, efficient high-performance interactive environment (the one desired in modern, real-life applications) the present situation must improve. Namely, a union of batch-job oriented, non-interactive, HPC environments and the components related to steering of these jobs and their settings during the run time – which used to be “contradictory” to each other – is nowadays an absolute imperative.

2.5 Visualisation

The visualisation component, without which intuitive data analysis and understanding would not be possible, must be attached to a user interface. Intuitive understanding and effective visual analysis is essential in the context of dynamic, interactive environments, where update rates should be high. Visualisation programs are preferably written for and executed on specialised, dedicated, efficient, powerful hardware – Graphics Processing Units (GPUs).

2.5.1 GPU Hardware

Pioneered in the late 1990s by 3dfx Interactive [9] and NVIDIA [10], GPU computing has quickly become an industry standard, enjoyed by millions of users worldwide and adopted by virtually all computing vendors [10].

Workstation graphics cards from AMD and NVIDIA are unbeatable for professional 3D visualisation. AMD’s FirePro and NVIDIA’s Quadro families have been competing for a long time for best hardware as well as the best driver support. NVIDIA’s Kepler is currently the world’s fastest and most efficient high performance computing (HPC) architecture [11]. In addition to 8 GB of memory with 320 GB/sec memory bandwidth and 3072 cores [11], it delivers more processing performance and efficiency through a new, innovative streaming multiprocessor design that allows a greater percentage of space to be applied to processing cores versus control logic (contrary to CPUs).

2.5.2 GPU Computing

The early issues from previous generation GPUs, related to cache coherence, or the possibility to compute only with single precision, have now been solved due to new innovative technologies. Nowadays it is possible to run not only visualisation, but also many simulation codes efficiently on this hardware. GPU computing utilises a GPU together with a CPU to accelerate general-purpose scientific and engineering applications [10].

Namely, a combination of CPU and GPU computing can be incredibly powerful, because CPUs consist of a few cores optimised for serial processing, on which serial portions of code may be run, while GPUs consist of many more smaller, but more efficient cores designed for extreme parallel performance of parallelised parts of the code.

From the program development point of view, the GPU programs use an extended C language, embedded in C/C++ host program. The data is loaded on an execute device, where computations are done and results are then copied to the host. The problem is split up into logical blocks of threads, all executing the same code on one microprocessor.

Still, it should be noted that any synchronisation, or data communication among threads causes overheads to be longer than desired. If the data, additionally, has to be copied many times to the CPU memory and back, the situation concerning latencies becomes even more dramatic. As a response to these issues, dynamic parallelism in the latest generations simplifies GPU programming by allowing programmers to easily accelerate all parallel nested loops – resulting in a GPU dynamically spawning new threads on its own without going back to the CPU [12]. On some architectures, such as NVIDIA's Kepler, multiple CPU cores are allowed to simultaneously utilise a single GPU, advancing programmability and efficiency.

To sum up, despite of all the advantages which GPU computing offers, some problems and the corresponding simulation codes are not highly concurrent by their nature in the way that is optimal for the execution on a GPU. These remain faster on CPUs. Many simulations still *have* to be written for the execution on CPU hardware, which the framework is suited and primarily aimed for. Furthermore, even if possible, it requires significant time and effort to port those application codes written already for CPU to their GPU counterparts.

Despite the wide application area (which can be found in the list on NVIDIA popular applications catalogue [13]), GPU hardware and computing are, within this work, discussed mostly with the intention to provide a more complete picture of hardware/software state-of-the-art in scientific computing. None of the simulation test scenarios for the framework run on GPUs, however, for some of the visualisation components, the acceleration due to GPU computing is significant. Due to this, acceleration is observed for the update rates of the whole interactive system as well.

2.5.3 Scientific Visualisation

"Visualisation is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualisation offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionising the way scientists do science." [123]

Today, scientific visualisation [92, 93, 83, 67] is still a key feature to foster insight into the complex scientific data. Interpretation of the huge data sets gained as a result of simulating complex phenomena is today unimaginable without it, whereas in addition to editing this data in a visualisable format, selective previewing, or lower accuracy previewing is possible to achieve interactive feedback rates. It maximises our capacity to perceive, analyse, and understand dynamic processes and situations in either theoretical, or fictitious, or the real world. The challenge in this field is certainly to efficiently visualise large-scale data sets and complex data structures.

In contrast to briefly presented simulation techniques, in scientific visualisation one normally tries to reconstruct a continuous real object from a given discrete representation, although there are also examples where data to be presented is discrete, such as some statistical values in geographical regions. Discrete structures consist of entities, from which grids/meshes consisting of cells are built. The continuous signal is known only at a few points (data points). In general, data is needed in between these points. By interpolation a representation that matches the function at the data points is obtained, which allows for constructing new values within the range of a discrete set of known data points.

As mentioned before, visualisation itself is not the focus of this work, however it is a necessary interactive computing component. Without real-time visualisation tools, there would be no possibility to intuitively interpret the results and guide the exploration of a computational model.

Scientific visualisation without integration into interactive computing environment may take e. g. image data, based on MRI/CT scans, digital landscapes, colour cryosections, etc. One famous example is the Visible Human Project [31], where bodies were frozen in a special material to preserve the tissues and organs. Sections were “shaved” off the frozen block in thin layers to expose underlying tissues and photograph them. The frozen section procedure is a laboratory procedure to perform rapid microscopic analysis of a specimen, the technical name for which is *cryosection* [25]. In this way, a “stack” of around two thousand two-dimensional cryosections were obtained (there are also MRI and CT sets) [31].

In an interactive computing environment, in particular, the data which is available for visualisation is either (1) directly a simulation result, i. e. the simulation component reads in the input data and the values of the parameters are influenced via a simulation component (e. g. a textual user interface), or (2) initial geometry/model data is read by a visualisation component, visualised and then passed to the simulation kernel for simulation. In the latter case, as soon as the results are received, further updates can be made via GUI, and the cycle repeats. This is described for the concrete examples in Chapter 4. It is always a good question which data to visualise (and how) and which to skip, so that the “important” data is most interpretable and not hidden by the information which is redundant or irrelevant for a particular purpose.

2.6 Software Engineering

In every engineering and scientific community, in interdisciplinary projects in particular, there is an urge for code designs which are easily extensible, modular and reusable, free for the whole community. Moreover, there are still open questions as to what kind of workflow is both scalable on different architectures, including massively parallel ones, and intuitive for the user. What kind of user interfaces should facilitate interactive scientific computing? Talking about "generic" concepts for every (or one common, "supersonic", next generation) interactive environment – the question remains if the whole, ever-growing diversity of applications would allow for anything generic, just like there are examples of industry APIs, without significantly compromising the performance.

In *software engineering*, a *design pattern* is a general reusable solution to a commonly occurring problem within a given context in software design [26]. A design pattern does not refer to the final code which can be integrated into an application, but rather a (reusable) formalised practice for solving a concrete problem, which is then to be implemented.

The way object-oriented design patterns show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved [26], many other patterns describe, e. g. application modules and their interconnections.

2.6.1 Design Patterns

The content presented at this point becomes relevant for this work when discussing integration of the framework into different application scenarios in Chapter 4. *SCIRun* uses a Model-View-Controller (MVC) design. Some codes, on the other hand, lack a formal design, which makes integration of the framework less intuitive.

The *Model-View-Controller* (MVC) pattern separates the simulated model from the representation of information and the user's interaction with it. The controller entity mediates all the input, informing the model or view component about the change. The central idea behind MVC is code reusability and separation of concerns.

The *Observer* pattern is another software design pattern in which an object, called the "observable", maintains a list of its dependents, called "observers", and notifies them automatically of any state changes, usually by invoking one of their methods. Observer is also a key part of the MVC architectural pattern.

2.6.2 Dataflow Model – Advantages for Interactive Steering

The dataflow model is a software architecture based on the idea that changing the value of a variable should automatically initiate (often a sequence of) recalculation

of other values which, indirectly or directly, depend on the initially changed one. This concept is utilised, for instance, in the *SCIRun* application scenario.

The obvious benefit of one such architecture for a highly interactive computing environment is that it can reduce the amount of coupling-related code in a program, i. e. that the update operation does not have to be explicitly contained in the program, and an explicit check doesn't have to be added to avoid cyclical dependencies.

2.7 State-of-the-Art Interactive Steering Environments

As mentioned earlier, the idea of interactive simulation steering has been present in the scientific and engineering community for more than two decades. Numerous powerful tools serving this purpose have been developed. In this section, an extended overview and classification of the state-of-the-art tools is provided, which the author already briefly discussed and which have been published in [105, 106, 102, 103, 108, 107, 109, 110, 104, 100]. The ideas behind the steering environments and systems such as CSE [165], VASE [88], Progress [169], Magellan [167], SCIRun [134, 136], Uintah [61], G-HLAM [148] and EPSN [132], libraries such as CUMULVS [77], or RealityGrid [53, 140, 20], frameworks such as Steereo [89], etc. are illustrated in order to be compared to the approach in this work.

2.7.1 Frameworks

Steereo [89] is a light-weight steering framework, where the client sends requests and the simulation side will respond to them. However, the requests are not processed immediately, but rather stored in a queue and executed at predefined points in the simulation. A developer must define in their code when this queue should be processed.

2.7.2 Toolkits

In *Progress* [169], steering acts on application level data structures. Actuators are function calls inserted by the user to instrument code at appropriate locations. Actuators are the mechanism through which steering actions are accomplished. They are placed where it would be safe to perform the steering action. The steering entities are registered within the steering server via function calls manually inserted into the application code.

Magellan's [167] architecture consists of a number of steering clients and servers. Steering servers can be executed as a separate thread in the application address space, they are modular, and can be adopted to a range of (e. g. parallel) application scenarios. Clients can control the steering system from various sources, including remote, heterogeneous sources, such as collaborative, immersive environments [168]. Steering objects are exported from an application. A collection of instrumentation points, such as so-called actuators, knows how to change an

object without disrupting application execution. Pending update requests are stored in a shared buffer until an application thread polls for them [167].

VASE (Visualisation and Simulation Steering Environment) [88] is designed for model studies and experimenting with algorithms. Steering is accomplished using programmer-defined breakpoints. One may alter the values of variables and parameters and add scripts and statements during program execution time. It is designed for a single parallel computing environment.

2.7.3 PSEs, Environments and Systems

CSE [165] is a computational steering environment consisting of a very simple, flexible, minimalistic kernel, and the modular components, so-called satellites, where all the higher level functionality is pushed. It is based on the idea of the central process, i. e. the Data Manager, to which all the satellites can be connected. Satellites can create and read/write variables, and they can subscribe to events, such as notifications of mutations of a particular variable [128]. The Data Manager informs all the satellites of changes made in the data and an interactive graphics editing tool allows users to bind data variables to user interface elements.

In the *SCIRun* [93, 134, 136, 95] problem solving environment (PSE) for modelling, simulation, and visualisation of scientific problems, a user may smoothly construct a network of required modules via a visual programming interface. Computer simulations can be then executed, controlled, and tuned interactively, triggering the re-execution only of the necessary modules, due to the dataflow model. This PSE has typically been adopted to support pure thread-based parallel simulations so far. *Uintah* [61] is a component-based visual PSE that builds upon the best features of the *SCIRun* PSE, specifically addressing the massively parallel computations on petascale computing platforms.

In the *G-HLAM* PSE [148], the focus is more on fault tolerance, i. e. monitoring and migration of the distributed federates. The main *G-HLAM* services consist of a coordinator which manages the simulation, a migration service which decides when performance of a federate is not satisfactory and migration is required, and another which stores information about the location of local services. It uses distributed federations on a Grid for the communication among simulation and visualisation components.

The *EPSN* API [132] is a distributed computational steering environment, in which an XML description of simulation scripts is introduced to manage data and concurrency at instrumentation points. There is a simple connection between the steering servers (i. e. simulations) and clients (i. e. user interfaces). When receiving requests, the server determines their date. The request is responded to as soon as it fulfills a condition. Reacting on a request means releasing the defined blocking points.

It is easy to construct PSEs in the *PSE Park* [111] framework. (In other words, based on the user's data, it outputs PSE). It supports program generation, easy modification of programs, PSE development, automatic documentation creation and simulation execution support via a Cloud.

2.7.4 Libraries

CUMULVS [77] is a (middleware) library that provides functionality so that a programmer can extract the data from a running (possibly parallel) simulation and send the simulation result data to the visualisation package. It encloses the connection and data protocols needed to attach multiple visualisation and steering components to a running application during the execution time. The user has to declare in the application which parameters are allowed to be modified or steered, or the rules for the decomposition of the parallel data, etc. Using check-pointing, the simulation can be restarted according to the new settings. CUMULVS is also collaborative, hence, a token scheme prevents conflicts among different users. Consistency protocols are used to verify that all the tasks in a distributed application apply the steering changes in union.

The *RealityGrid* [53, 140, 20] project has provided a highly flexible and robust computing infrastructure for supporting the modelling of complex systems [20]. An application is structured into a client, a simulation, and a visualisation unit communicating via calls to the steering library functions. This infrastructure involves insertion of check- and break-points at fixed places in the code where modified parameters are obtained and the simulation must be restarted. It represents an example of job- and application-level steering.

The *Pablo* [146] library routines are aimed at instrumenting the source code. It is intended to extract performance data as the code executes. It utilises sensors to collect the information from the executing code and actuators to do the alterations. It does performance steering, varying cache size and cache block replacement policy as needed.

2.7.5 Collaborative Steering

As mentioned in Chapter 1, in collaborative environments *multiple users*, e. g. experts in different fields, can attach to the application and steer it simultaneously. One of the reasons why desktop and application sharing can be unsuitable is according to [47] "the fact that specialists from different domains make different demands on the application they are using while taking part in the collaborative session".

The *Cactus* [34] simulation framework provides a generic code module (a "thorn") which implements a web server within a system for collaborative research. It uses standard Unix socket communication calls on the network I/O layer. It listens on the chosen port to the HTTP client request. Within this astrophysics community framework, new computing technologies have been developed, which

allow massively parallel simulation codes to be simultaneously developed and used by multidisciplinary community [34].

DISCOVER (Distributed Interactive Steering and Collaborative Visualization Environment) [120] is an environment for web-based, collaborative interaction of scientists and engineers with a running simulation. It consists of detachable thin clients at the front end, a network of interaction servers (which are web servers, extended to handle real-time simulation results and user request data) in the middle, and a set of sensors, actuators and interaction agents at the back end [120]. Clients (users) can interact with registered applications using a web browser.

Distributed laboratories [141] provide application-specific event-based monitoring of parallel and distributed applications. The aim is to explore highly distributed and dynamic target platforms. The resulting set of values is forwarded to the computational instrument via the steering infrastructure and is available as a model in the next timestep. At any point the user can checkpoint the application execution by pressing the button or invoke a default checkpointing policy which automatically saves the application execution history after a predefined number of timesteps. The various programs that make up Distributed Laboratories need not all be under the control of single group, compiled by the same compiler or even written in the same language.

COVISE (COllaborative VIsualization and Simulation Environment) [14] is a software environment which integrates simulations, postprocessing and visualisation functionalities. It is easily extendable and supports remote distributed computing on heterogeneous machines. An application consists of several modules, i. e. processes. Each processes can be arbitrarily spread across different platforms. The users can collaboratively analyse the simulation result in a fully immersive Augmented Reality environment. One of the users (the “master” user) can add others to the collaborative session.

Many of these powerful tools have targeted a particular application only (to the needs of which they are adapted, and perform according to expectations), however, they are not applicable to many other application scenarios. Others involve major simulation code changes in order to be effective for different problem sizes (and, independent from the size, apply an optimal check-pointing strategy for recognising user updates). This makes scientific and engineering communities reluctant to use them. The vision inspired by all the good properties of the state-of-the-art tools leads naturally to the implementation of a software which can be used in a wide spectrum of engineering codes, without requiring major code changes.

2.8 Projects within the Chair for Computation in Engineering

Within the Chair for Computation in Engineering at the Technische Universität München, a series of successful computational steering research projects took place in the previous decade. Industry partners have also been involved. Performance analysis has been done for several interactive applications, in regards to their responsiveness to steering, and the factors limiting performance have been successively identified. The focus at this time was namely on interactive computational fluid dynamics (CFD), based on the Lattice-Boltzmann method [115], including Heating Ventilation Air-Conditioning (HVAC) system simulator [48], online-CFD simulation of turbulent indoor flow in CAD-generated virtual rooms [175], interactive thermal comfort assessment [166, 138], and also on structural mechanics – namely, computational methods in orthopaedics [178].

In online-CFD simulation of turbulent indoor flow for CAD-generated virtual rooms, the underlying CFD parallel simulation is processed on a high performance supercomputer based on master-slave strategy. The propagation steps' data among the slaves is exchanged via vendor optimised MPI. Since the supercomputer is not the optimal place for the visualisation component to run, the visualisation is done on the workstation. To handle the huge 3D data set more efficiently a VR environment is integrated into the computational steering concept [175]. The transformation of objects is performed by a wand device interacting with draggers [175]. The visualisation of cutting planes, streamlines, vector planes, etc. are provided by an extension of the TGS Open Inventor library. The coupling of the simulation with the VR environment is done by an MPI distribution which supports heterogeneous environment. To make this communication efficient, the steering information is sent only when the user interacts and the simulation results are transmitted in regular, user-predefined intervals. When the user interacts, he gets feedback on the most recent result, right after the interaction.

In the Collaborative Computational Steering Platform (*CoCoS*) [48], geographically distributed engineers can collaborate on solving the same problem numerically. The implementation assumes a central server component which holds the 3D geometric model data, boundary conditions, machine and material parameters, etc., which can be accessed by multiple clients in a consistent way. Due to the locking-based, centralised approach, conflicts cannot occur. For the client component, logging in/out of a server is possible, as well as visualisation and modification mechanisms for the geometry, display of additional data related to the selected object and overview of other "online" users.

In [166, 138] the air-flow is examined interactively, but more in the context of thermal comfort assessment. A VR user interface for visualisation and user interaction forms a front-end, while on the back-end a parallel CFD kernel and a 3D grid generation component are running. User interaction, depending on the type of interaction, can be performed via providing data in input files, or via dialogs, or even via VR devices, such as draggers. In the collaborative version

of the project, multiple clients connect to a common server containing geometry data using the Orbacus CORBA [15] communication library. This way the MPI compatibility issue among different platforms is elegantly solved.

Over time, valuable observations and experience have resulted in significant reduction of the work required to extend an existing application code for steering. Nevertheless, the developed concepts have been primarily adapted to this limited number of application scenarios. They allow for further investigations so as to become more applicable – in the same time efficient enough, generic, and easy to implement. This is where the new concept comes into play.

2.9 Difference of our Approach

An integration framework applicable to different engineering applications is introduced, which – with only minor code modifications involved – enables real-time interactive computing. Cases of parallel simulations (with shared and/or distributed memory) are supported, as well as visualisation on-the-fly. For interrupting the regular course of a simulation, software equivalents of hardware interrupts are used. This way, the control is shifted from the simulation code to the operating system. Checking for the updates is based on a generic mechanism, involving user defined intervals instead of the “smart” placement of check-points in the code. For the communication of the simulation back-end and GUI/visualisation front-end typically non-blocking asynchronous communication routines are used, or such communication is simulated via multithreading mechanisms. Different applications in terms of algorithm and code structures, in terms of time and memory consumption, parallelisation techniques, and engineering field have been tested; different programming languages, compilers and operating systems are supported, as detailed in Chapter 3.

2.10 Conclusions

Efficient simulation is absolutely necessary as a basis component in an interactive process. For large scale simulations, it may be advantageous to use approximation methods and solvers based on hierarchies, i. e. adaptive meshes, *p*-FEM “natural” hierarchy [145, 144, 73, 131, 19, 159], sparse grids [55, 59], efficient multigrid solvers [50, 82, 51, 163], etc. When simulation memory requirements exceed the available resources, out-of-core methods must be employed. A survey of general out-of-core algorithms for linear systems can be found in [162]. The solution vectors, however, often have to be kept in main memory, which is sometimes not possible. Therefore, more sophisticated ordering methods are needed for accessing these large data. To achieve a fast interactive trial-and-error process needed for intuitive parameter tuning, parallelisation methods are desired on many of the application levels.

What is required for an effective parallel computation approach is the following combination of hardware and software capabilities [101]:

– The interconnection between processors and memory must allow rapid communication between the individual processes as well as fast transfer of data to and from memory.

– A protocol must be available for the communication between processes – either based on ports (channels) or on process identifiers. Improving the network architecture is critical, however, research must offer techniques to reduce the demand on the network [33] between all the application components.

– It must be possible to effectively separate the computational algorithms and input data into individual sub-problems (the decomposition issue).

– Evidently, it must be possible to assign individual sub-problems to separate processes.

– Load balancing for the system – dividing the required work equally among all of the available processes, which ensures that one or more processes do not remain idle while the other processes are actively working on their assigned tasks. In this way, valuable computational resources are not wasted.

Scientific visualisation is crucial for intuitive interpretation of results. In interactive computing it is also essential to achieve real-time update rates. Thus, depending on the application, it might be advantageous to do selective updates of only part of the domain which is of interest, with adapted resolution, etc. Otherwise, for large data sets, updates would not be possible for the whole domain within real-time interactive rates.

It is very important to have powerful hardware available, however, to exploit it seems to be more challenging. Moore's law in software technology is the main prerequisite for exploiting upcoming exascale architectures.

As the last conclusion of this chapter, the author finds nothing more adequate than a quote from the 2011 report about researchers presenting experiences from ENZO, CACTUS, and iPlant API development efforts [23]: "Nearly every field of science has a community code (or several) that satisfies a large percentage of the discipline's scientific needs. Great minds – and thousands of hours of PhD and post-doc labor – have gone into the creation of these codes. However, as new technologies emerge that are capable of delivering millions of times the power as previous systems, it is often necessary to rethink and rewrite these existing community codes, which is no small step."

Chapter 3

Implementation

In interactive computing environments, it is essential that no component, which has to be re-executed when an update occurs, requires so much time that the connection between the cause, i. e. the update, and the corresponding effect is lost. The frequency of the updates perceived by a user during interactive exploration of a computational model depends on the component that requires the longest amount of time to process at least one update [44], i. e. one iteration. There are various challenges which have to be faced in order to interactively steer a long-running simulation, therefore achieving the corresponding updates in real-time. As said before, a long-running simulation, as defined in this dissertation, assumes one with execution time ranging from a second to a day, or longer – anything which is not real-time.

First of all, a simulation program, running often remotely on a high performance cluster, should immediately become aware of the user update, skip the outdated computation, and start anew. However, the rate of checking for updates should not dramatically affect the execution time of a simulation program.

Another issue is how to develop a relatively generic concept, one which can be used for various simulation scenarios, and how can users with various requirements profit from that concept. How can users avoid, for instance, placing checkpoints in the code based either on experience of previous runs, or *a priori* estimations of the execution time of one iteration?

Finally, in the case of large data sets which need to be transferred as a simulation result, the challenge of transferring and editing this result in real time remains. The implemented interactive computing framework tackles all these issues.

3.1 The Concept of the Framework

The first intention was to follow the minimal invasion principle and, thus, shift potential changes of an initial code as much as possible towards the integration framework functionality.

In addition to this, the responsiveness of a simulation to changes should depend more on the user-predefined settings (how often the update check is desired), than on the time which e. g. one iteration takes.

One of the basic options is certainly to place check-points at pre-defined places in the code. A simple example should be enough to illustrate the major issues of this approach. Let us consider an iterative function with several nested loops being executed, whereas users can interact at any point, requesting some update of the data. Obviously, the difficulty has to be confronted – *where* within the given code would be the optimal place for check points, so that within the function the updates are definitely applied as soon as possible. If they are placed inside the *innermost* loop, checking is presumably frequent enough, i. e. so that the relation between an update and a response of the simulation remains intuitive for a user. This, however, might cause significant time overheads, especially when communication library calls have to be employed to do the aforementioned checks, i. e. to *probe* a message. Checks in the *outermost* loop might turn out to be not frequent enough, i. e. the simulation is not responding in real-time to the actual user demand (see Fig. 3.1, left).

Another option would be to employ one thread per process to wait and check for updates (see Fig. 3.1, right). Here, both of the scenarios have to be taken into account – the one where a user process and a simulation process have access to a common, shared memory, and the distributed scenario, i. e. where (a) simulation process(es) are informed about user updates via a message passing interface. In either case, if not implemented carefully, computational resources, especially if they are very limited, may be wasted. To prevent this, setting the checking thread to *sleep* or *yield* periodically is mandatory, thus, releasing the occupied resources, as long as there is no update (a shared variable update) initiated by a user. In the MPI case, the solution would most likely involve setting the MPI variable MP_WAIT_MODE to *nopoll*, by which the thread waiting for the message would keep polling the message dispatcher for a very short time (less than 1 millisecond, according to the IBM MPI Programming Guide [29]). Then, it would release the resource it is running on until either an interrupt occurs, or time expires.

The idea of the framework is conceptually similar to the latter option described. However, speaking of implementing and integrating a generic concept, it would need to predict a variety of possible threading APIs and scenarios on the simulation side. Moreover, that implementation concept would certainly result in the necessity to re-structure and/or re-write parts of the initial code. This would mean either: (1) introducing multithreading in originally sequential codes, or (2) in codes originally having multithreading enabled – modifying existing implementations of parallel regions, due to possibly conflicting APIs of these codes and the framework itself.

Therefore, a new concept is developed as a part of this work. It overcomes the aforementioned potential issues using software equivalents of hardware interrupts, i. e. *signals*. Thus, a brief overview of this concept is provided – first of all for Unix

Within an iterative function:

for (i ← 0 to N) do
    Check point here?
    for (j ← 0 to M) do
        Check point here?
        Compute (data[i][j])
    od
od

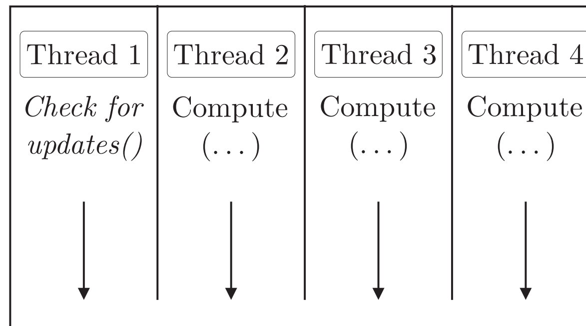


Figure 3.1: Checks for updates – left: insertion of check-points in code; right: dedicating one thread per process to do only checks for updates. Having in mind a generic concept of the interactive computing framework, it is rather difficult to predict the threading API used in each application scenario. However, this would be necessary in order to avoid conflicts of using two different APIs in one code.

systems, whereas further in this subsection portability to Windows operating systems is considered as well.

3.1.1 Signals Introduction

Signals are software interrupts [152]. Applications generate signals to inform about their situation [152], or users may generate them by pressing certain combinations of terminal keys; hardware exceptions, such as invalid memory reference, can be a reason for the corresponding signal to occur. More details on when a signal may occur, or how to invoke it, is provided further in the text.

Linux supports both POSIX reliable signals ("standard signals") and POSIX real-time signals. Each signal has a current defined action, which determines how the process behaves when the signal is delivered. Default actions, according to Linux Manual pages [27] can be to terminate the process, to ignore the signal, to terminate the process and dump core, stop the process, or to continue the process if it is currently stopped.

A programmer may change the action of a signal using the *sigaction* system call in his code, or (its less portable counterpart) *signal* (Code 3.1). Using these system calls, an active process can elect on delivery of the corresponding signal whether it is performing the *default action*, *ignoring* the signal, or *catching* the signal with a *signal-handler*. A signal-handler is a programmer-defined function that overrides the default behaviour, i. e. *disposition* or *action*, on signal delivery.

sigaction provides more control, allowing for specifying additional control flags. The structure sigaction, used to describe an action to be taken, is defined in the header <signal.h> and includes at least the members listed in Code 3.1.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int signum, sighandler_t handler);

#include <signal.h>
int sigaction (int sig, const struct sigaction *act,
               struct sigaction *oact);

struct sigaction members:
void(*) (int) sa_handler – SIG_DFL, SIG_IGN or pointer to a signal-handler,
sigset_t sa_mask – signals to be blocked during execution of signal-handler,
int sa_flags – special flags to affect behaviour of signal,
void(*) (int, siginfo_t *, void *) sa_sigaction – signal-handler.
```

Code 3.1: Signal and *sigaction* specification.

A signal may be *blocked*, which means that it must not be delivered at that time point. Between the point when it is generated and the point it is delivered, a signal is said to be *pending*. It is delivered later, as soon as it is *unblocked*. The list of blocked signals is maintained within the *signal mask*.

Raising signals

On UNIX based systems, signals can be raised in one of the following ways:

- *alarm(unsigned int t)* – which arranges SIGALRM signal to be delivered to the current process in *t* seconds; *ualarm(t)* is its equivalent, whereas an interval *t* is in microseconds;
- *settimer* – (conforming to SVr4, 4.4BSD) similar to, but more powerful than *alarm*; it is, however, used less often due to the more complex semantics (the synopsis is provided in Code 3.2);
- keyboard taster combinations: *CTRL-C* sends SIGINT, *CTRL-Z* sends TSTP signal (SIGTSTP), and *CTRL-* sends a QUIT signal (these default combinations can be modified by *stty* command);
- *kill* system call/command sends the specified signal to a process if permissions allow, such as termination of the process, e. g.;
- *raise* library function sends the specified signal to the current process;
- *abort* function causes abnormal process termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return [81];
- *pthread_kill* upon successful completion arranges that a signal is delivered to the specified thread;

```
int setitimer ( int which, struct itimerval * new_interval,
                struct itimerval old_interval);

/* Timer values are defined by the following structures: */
struct itimerval {
                struct timeval it_interval; // next value
                struct timeval it_value; // current value
}

struct timeval {
                long tv_sec; // seconds
                long tv_usec; // microseconds
}

There are three interval timers:
ITIMER_REAL – provides SIGALRM signal (real-time timer)
ITIMER_VIRTUAL – provides SIGVTALRM signal (virtual time timer)
ITIMER_PROF – provides SIGPROF signal (profiling timer)
```

Code 3.2: *settimer* synopsis. *settimer* notifies of a timeout according to the chosen interval timer (real, virtual or profiling). When the parameter *new_interval* is nonzero, an appropriate signal is invoked in the specified interval. When the *old_interval* is nonzero, the old value of the timer is stored there.

- *sigqueue* provides signal specified to be sent and *queued* with the specified value, which is discussed in more detail later in this chapter.

Standard signals

As mentioned before, Linux supports standard signals. A reader may note that several signal numbers are architecture dependent. Tab. 3.1 contains a list of some of the signals described in the original POSIX.1-1990 standard.

Some signals, such as SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Code 3.3 shows first examples of code segments using signals (in C programming language). The provided “solutions” use either *sigaction* or *signal*. However, *sigaction* is favourised, since it is not specified whether *signal* automatically resets the signal handler. This means that *signal* needs to be explicitly called again inside the handler. Furthermore, if there are two signals in quick succession, and the second is delivered before the handler is reinstalled, the default action will be applied. This is often to terminate the receiving process. *sigaction*, on the other hand, is guaranteed *not* to reset the defined signal action.

Signal	Number	Action	Summary
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Timer signal from alarm
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCONT	19,18,25	Cont	Continue if stopped
SIGTSTP	20	Stop	Stop process, can be ignored/caught
SIGSTOP	17,19,23	Stop	Stop process

Table 3.1: Unix signals.

Nevertheless, a developer should be aware that *sigaction* might be unavailable on some Linux systems, since *sigaction* is conforming to POSIX standard and *signal* to POSIX, ANSI C 89 and 99. For *sigaction* to be supported, Linux test macro requirements which have to be fulfilled for GNU C Library are either _POSIX_C_SOURCE ≤ 1, or that either _X_OPEN_SOURCE or _POSIX_SOURCE is defined.

There are also other functions similar to *signal* and *sigaction*, such as *bsd_signal* and *sysv_signal*. However, just like *signal*, they are not recommended [22, 27]. This is due to their often even more serious reliability and portability issues. To sum up, one should choose *sigaction* whenever possible and, thus, it is used in all further examples.

```
sig_atomic_t volatile keep_computing; /* globally visible var. */

/* The signal handler just clears the flag and re-enables itself */
void handle_alarm (int sig)
{
  keep_computing = 0;
  signal (sig, handle_alarm);
}

// Within the main function:
/* Establish a handler for SIGALRM signals */
signal (SIGALRM, catch_alarm);

/* a better alternative would be:
struct sigaction new_action;
new_action.sa_handler = handle_alarm;
sigemptyset (&new_action.sa_mask);
new_action.sa_flags = 0;
sigaction (SIGALRM, &new_action, NULL); */

/* Set an alarm to go off in a little while */
alarm (2);
/* Check the flag once in a while to see when to quit */
while (keep_computing)
      do_computation (); /* a desired function */
```

Code 3.3: Catching ALARM signal, using signal and sigaction

Real-time signals

Apart from standard signals, Linux supports 32 real-time ones, as included in POSIX.1-2001. They are numbered typically from 32 (SIGRTMIN) to 63 (SIGRTMAX). Unlike standard signals, the entire set of real-time signals can

```
#include <sys/types.h>
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value);
```

Code 3.4: Synopsis sigqueue

be used for application-defined purposes, except from the first three real-time signals, which are used by LinuxThreads. The default action for an unhandled real-time signal is to terminate the receiving process.

If multiple instances of a standard signal are delivered while that signal is currently blocked, only one instance is queued. By contrast, multiple instances of real-time signals *can* be queued.

A real-time signal must be invoked using *sigqueue* (synopsis is provided in Code 3.4) in order to queue all the upcoming signals. Here, one can send also an accompanying value (either an integer or a pointer) with the signal. If the receiving process establishes a handler for this signal using the SA_SIGINFO flag to *sigaction*, then it can obtain this data via the si_value field of the siginfo_t structure passed as the second argument to the handler. The si_pid and si_uid fields of this structure (which are omitted in Code 3.1) can be used to obtain, respectively, the process identification and the real user identification of the process sending the signal.

Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal (i. e., low-numbered signals have highest priority).

Different compilers

POSIX standard says for the usage of *sigaction* that the action for a signal need not be refreshed after signal invocation in order to have the next signal handled as desired. However, the experience with some compilers (Intel, e. g.) shows the opposite. Sometimes it has to be explicitly "required" that the signal action is refreshed before the next signal occurs.

Different programming languages

The previous examples referred to the signal handling within C/C++ codes. In order to make the framework applicable to Fortran codes as well, support for signal handling in this programming language has to be considered.

Signals with Fortran

On the one hand, support for signal handling in Fortran can be achieved at user level with minimal efforts. Some vendor supplied Fortran implementations, including for example Digital, IBM and Intel, have the extension that allows the

```
// The code in a C-file:
typedef void (*sighandler_t)(int);
/* "clean-up" wrapper in C */
void sigclear_(int *signum)
{
  signal (*signum, NULL);
}

/* wrapper for the signal in C */
void signal_(int *signum, sighandler_t handler)
{
  signal(*signum, handler);
}

! Fortran code segments:
subroutine handle_alarm
            do_computation ()
end subroutine handle_alarm

! Within the main function:
call sigclear(2) ! "clean-up" for signal(2) = SIGINT
call signal(2, handle_alarm) ! set the new signal(2) behaviour
```

Code 3.5: Signals in Fortran

user to do signal handling as in C [43], thus, a C wrapper function for overriding the default signal behaviour is implemented.

On the other hand, one has to take into consideration that the behaviour of the corresponding Fortran function is implementation dependent. GNU Fortran has defined one such as an intrinsic function, i. e. a compiler makes it available for use. If one uses Intel Fortran compiler, the specified signal handler function does not provide the expected result when the corresponding signal occurs. In order to take actions defined in the signal handler one needs to "clear" the previously defined action first (Code 3.5).

Windows OS signals

Finally, it is important to discuss portability issues to Windows operating systems. Windows is an event-driven operating system, thus, only the timers provided by Windows can be used. In other words, hardware interrupts cannot be invoked for timing purposes directly in an application (unless the application is a device driver itself, e. g.). There are two basic types of timers for Windows:

- the system timer,
- the multimedia timer.

The concept of timers is similar to the one in Unix systems – a time interval for the timer is specified and the timer is activated to fire each time the time period expires. In the Windows systems, however, a *callback* function, which will be called each time the timer fires, has to be specified for the timer. Optionally, in the case of the Windows system timer, WM_TIMER message can be sent to the application's Window procedure each time the timer fires.

The system timer is according to [147] easiest to use, but less accurate, as the lowest interval one can effectively use is about 55 milliseconds. It is also less reliable than the multimedia timer. The reliability issue is related to the mechanism of the WM_TIMER message to notify a user each time the specified interval elapses and the priority of this message. The other way to create and use a system timer is using the Win32 API – the *SetTimer* and *KillTimer* functions, or VCL *TTimer* component. In the latter case, an interval is specified (in milliseconds) and an event handler for the *OnTimer* event has to be implemented.

Of all the messages in Windows, WM_TIMER is one of the lowest priority messages. Thus, if the system is busy, WM_TIMER messages may be delayed, or removed from the message queue altogether. This would not be a major issue concerning the framework itself, since, as described in the next subsection, the timer would be set to fire periodically, thus a message that something has changed on the user side would be definitely received at some point. Nevertheless, since the response of the simulation would be limited to the 55 milliseconds interval, this method is not considered for the framework implementation.

A solution to reliability and accuracy issues of the system timer is provided by the Windows multimedia timer. Despite of its name implying a particular purpose, the multimedia timer is not limited to multimedia. It has a resolution of 1 millisecond. Furthermore, it is reliable, i. e. not the subject to the lost messages.

The multimedia timer functions are found in the MMSYSTEM.H header. This header need not be explicitly included, depending on the version of C++Builder used. To use the multimedia timer, the minimal time resolution needed has to be set via *timeBeginPeriod*. Next the *timeSetEvent* is called (see Code 3.6). *timeSetEvent* returns zero if an error has occurred, or a timer handle if the function has succeeded. Each call to *timeBeginPeriod* must have a matching call to *timeEndPeriod*, passing the same values.

The Windows documentation for the callback function says that only a few functions such as the *PostMessage* can be called from a callback function. Thus, a user-defined message (and its handler) can be created in the application's main form class. A message is posted by calling *PostMessage* from within the callback. This way the handler for the user-defined message does all the work.

Regardless of the timer used, it has to be ensured that the code that executes as the result of a timer event is short and concise. This holds especially if the timer interval is short. Namely, all the timer event instructions have to be executed before the next timer event occurs. A developer who sets the timer interval to one millisecond, for instance, should know how much time is needed for the

```
int time_int, delay;
timeBeginPeriod (1);
MMRESULT timerID;
...
timerID = timeSetEvent (time_int, delay, TimeProc, 0, TIME_PERIODIC);
/* timeSetEvent parameters:
1. parameter: time_int – the timer interval,
2. parameter: delay – delay (zero means demand for the highest possible accuracy),
3. parameter: TimeProc – used to pass the address of the callback function,
4. parameter: used to pass user-specified data (or a pointer to it),
5. parameter: the type of the timer: TIME_ONESHOT (fires one time), and
              TIME_PERIODIC (the timer fires repeatedly, in cyclic intervals). */

The signature of the callback function:
(must be a stand-alone and not class-member function)
void CALLBACK TimeProc (
                          UINT uID, UINT uMsg, DWORD dwUser,
                          DWORD dw1, DWORD dw2);
/* uID – timer ID of the timer that generated the event,
uMsg – reserved by Windows and is not used,
dwUser – used to pass user-defined data to the callback (the data passed in
         the call to timeSetEvent) will be passed to the callback function in
         this parameter,
dw1, dw2 – also reserved and should not be used. */
```

Code 3.6: Multimedia timer functions.

```
int i, j;

void CALLBACK TimerProc(UINT uTimerID, UINT uMsg,
                        DWORD_PTR dwUser, DWORD_PTR dw1,
                        DWORD_PTR dw2)
{
  some_action(i, j); // might need to be called via PostMessage
}

int _tmain()
{
  timeBeginPeriod (1);
  MMRESULT mmr = timeBeginPeriod(1);
  mmr = timeSetEvent (1000, 0, &TimerProc, 42, TIME_PERIODIC);
  for (i = 0; i < Imax; i++) {
      for (j = 0; j < Jmax; j++) {
          do_computation (i, j);
      }
  }
}
```

Code 3.7: Signal handling in Windows OS.

aforementioned instructions. A good profiler is indispensable in determining this. TurboPower's Sleuth QA Suite [30] is suggested in [147].

3.1.2 The Concept of the Framework

In order to leverage interactive computing for a broader scope of applications, the immediate response of the simulation side to the changes made by the user is indispensable. The major steps which are taken in order to achieve this goal are the following:

- interrupting the simulation to check for updates,
- restarting the desired (part of the) computation automatically and instantly if an update has occurred,
- re-computation,
- sending simulation results to the user for (visual) interpretation.

Interrupting the simulation

If coupled to our software, the regular course of the C/C++/Fortran simulation is being interrupted in small, cyclic intervals ensued by a check for updates. For this, signal SIGALRM would be invoked and caught in predefined intervals.

```
sig_atomic_t volatile keep_computing = 1; /* globally visible var. */

/* The signal handler just clears the flag */
void catch_sigint (int sig)
{
  keep_computing = 0;
}

// Within the main function:
/* Establish a handler for SIGINT signals */
struct sigaction new_action;
new_action.sa_handler = catch_sigint;
sigemptyset (&new_action.sa_mask);
new_action.sa_flags = 0;
sigaction (SIGINT, &new_action, NULL);

/* Check the flag once in a while to see when to quit */
while (keep_computing)
       do_computation();
```

Code 3.8: Catching SIGINT signal

Optionally, one may choose not to invoke automatic, periodic alarm, but do checks for updates on user demand instead. A user may demand a check for updates by sending SIGINT signal to the simulation back-end via pressing a combination of keyboard tasters, as described before. An example of handling this signal is presented in Code 3.8.

If a check "proves" that there has been no user activity, the control is given back to the computation, which continues from the previous interrupt-point either until the stage when the results should be sent to the user process, or until the expiration of another time interval.

Otherwise, the new data is received. The receipt of an update message is considered to be instantaneous. As the next step, the update of the old data is required. However, it is the responsibility of the user himself to instruct the simulation program, before compiling and running it, to match the received data to the simulation-specific requisites.

Restarting the computation

After an update has arrived, the computation is intended to be restarted using one of the supported concepts: (1) (sig)setjmp and (sig)longjmp, or (2) manipulating simulation-specific variables in the signal handler.

```
#include <setjmp.h>
void longjmp (jmp_buf env, int val);
void siglongjmp (sigjmp_buf env, int val);

/* sigsetjmp – set jump point for a non-local goto */
#include <setjmp.h>
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savemask);
```

Code 3.9: (Sig)setjmp and (sig)longjmp synopsis.

(Sig)setjmp and (sig)longjmp

longjmp, *siglongjmp*, *setjmp* and *sigsetjmp* (see Code 3.9 and Algorithm 3) are primarily used for handling errors and interrupts.

Both *setjmp* and *sigsetjmp* call save their calling environment in their *env* argument for later use by *longjmp* or *siglongjmp* respectively. *longjmp*/*siglongjmp* call restores the environment saved by the last call of *setjmp*/*setlongjmp* in its *env* argument. *siglongjmp* is similar to *longjmp* except if *sigsetjmp* used a nonzero flag as a second argument, when *siglongjmp* also restores the set of blocked signals.

(sig)setjmp returns 0, if the return is from a successful direct invocation. If, however, the return is from a call to *(sig)longjmp*, *(sig)setjmp* returns a non-zero value *val*. This allows for recognising when the return is from the signal handler due to arrival of a user update, and treating this case independently from a direct invocation.

If the value of the *savemask* argument is not 0, *sigsetjmp* will also save the current signal mask of the calling thread as part of the calling environment, so that it can be restored after a next user update, thus, the next invocation of this function.

All accessible objects have values as of the time *siglongjmp* was called. The exception might be the objects of automatic storage duration, which are local to the function containing the invocation of the corresponding *sigsetjmp*, unless they have volatile-qualified type (which is introduced later in the text). Namely, their values are indeterminate if they are changed between the *sigsetjmp* invocation and *siglongjmp* call [81].

This approach is most generic, since the application code can be used as a black box. Some simulation specific reinitialisations can even be implemented within the signal handler if needed.

Manipulating variables in the signal handler

If one chooses the concept of manipulating simulation-specific variables, a simulation code is not recognised as a black-box anymore. Namely, there has to

Algorithm 3 Longjmp within the handler.

```
1:  jmp_buffer env
2:
3:  signal_handler_code (int p):
4:  siglongjmp(env, p)
5:
6:  // Within the main function:
7:  for (t ← T_0 to T_N) // iterations over time
    do
8:    if (sigsetjmp(env, savemask=1) != 0) then
9:      do_reinitialisation()
10:   end if
11:   for (id_x1 ← X_10 to X_1N) interval do
12:     for (id_x2 ← X_20 to X_2N) do
13:       // Can be interrupted at any point,
14:       // id_xi are reset in the interrupt handler:
15:       process(data[id_x1][id_x2])
16:     end for
17:   end for
18: end for
```

be at least one interface (simulation) variable registered within the framework. According to some of the standards of multithreading libraries (OpenMP, e. g.), branching (i. e. jumping) out of the parallel region is not allowed. Therefore, this approach becomes inevitable in the multithreaded scenario. The section related to the integration of the framework into multithreaded codes discusses this in more detail. Nevertheless, the approach is illustrated in the context of a sequential code scenario as well, for the sake of completeness.

As an illustrative example, let us take a code which consists of several nested loops. The idea is that the computation is restarted, as a response to an update, by manipulating the iteration vector $i = (id_{x_1}, id_{x_2}, ...id_{x_n})$, i. e. the loop indices id_{x_i} of all the loops. This is done by setting i for each loop index to the predetermined maximal value, as shown in Algorithm 4.

In the sequential scenario, there is no difference which variables are manipulated. When the multithreading comes into play, the private copies of i – belonging to the corresponding threads – are not visible inside the signal handler.

Re-computation

In the first approach, i. e. *sigjmp* example, the restart of the computation is immediate, and the restart point is the place in the code which was executed when *setjmp* was called previously.

In the second approach, when the control of the execution is given back to the main computation, it is obliged to continue at the point where it has previously

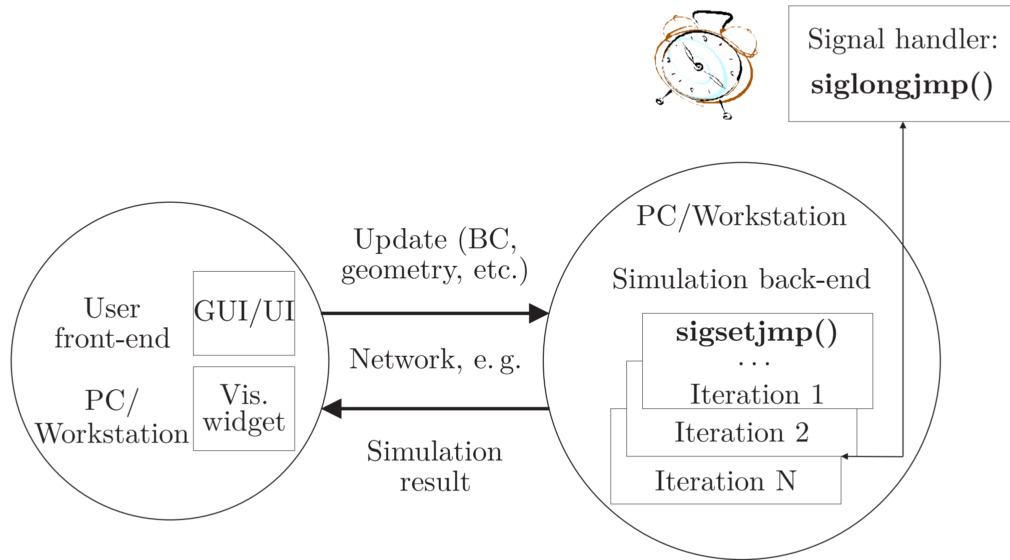


Figure 3.2: The framework with sigsetjmp, siglongjmp.

been interrupted. Conveniently, this happens only until the end of current, innermost loop iteration, where the earliest opportunity is used to compare the current value of the loop index with maximum value and analogously exit all the loops (i. e. starting with innermost and finishing with the outermost one). In the next steps of the algorithm (i. e. a new iteration), all the loop indexes are re-initialised with zero or some other simulation-specific initial value, thus, the computation is resumed. All the necessary reinitialisation routines are well supposed to be executed before this point, in order to guarantee the correct new computation. To guarantee the correct execution of the program, a developer has to be aware of at least a few important matters when handling interrupts, as discussed in Subsection 3.1.3.

Sending a result

When one or several iterations have been finished without an interrupt, the new results are copied into the send buffer and forwarded to the user. Again, it is then the *user's* responsibility to instruct the program, this time the one running at the front-end, how to interpret the received data correctly so that it can be visualised.

3.1.3 Challenges

When employing signals within a software implementation, many factors have to be taken care of – from interactions among different signals, to atomicity of operations, memory consistency, deallocating memory which will no longer be referenced, interaction of *alarm* with other function calls, etc.

Algorithm 4 Manipulating loop indexes within a handler

```
1: id_x1, id_xn declared global, sig_atomic_t, volatile
2: alarm_signal_handler():
3: manipulate id_x1, ... id_xn
4: reset_alarm()
5:
6: Within the main function:
7: init_sigaction()
8: alarm (interval) // set alarm
9: for (t ← T_0 to T_N) // iterations over time  do
10:    for (id_x1 ← X_10 to X_1N) do
11:       for (id_x2 ← X_20 to X_2N ) do
12:          // can be interrupted at any point, id_xi are
13:          // reset in the interrupt handler
14:          process(data[id_x1][id_x2])
15:       end for
16:    end for
17: end for
```

Interaction among different library calls

According to GNU C Manual [28], each process has three independent interval timers available:

- a real-time timer that counts elapsed time. The timer sends a SIGALRM to the process when it expires,
- a virtual timer that counts only processor time used by the process. This timer sends a SIGVTALRM signal to the process when it expires,
- a profiling timer that counts both processor time used by the process, and processor time spent in system calls *on behalf* of the process. This timer sends a SIGPROF signal to the process when it expires.

Theoretically, when *settimer* is supported on a system, any of these timers could be used. The synopsis can be found in Code 3.2 and an implementation example in Code 3.10 at the end of this section. Still, there can be only one timer of each type set at the time. Hence, a potential use of the same timer must be compromised in the initial simulation code (if planned to be executed in conjunction with the framework). On the systems where *settimer* is not supported, the less powerful *alarm* is used instead.

Moreover, interactions between *alarm* and any of *settimer*, *ualarm*, or *usleep* are unspecified [22]. Thus, they are not desirable within the initial code.

If *signal* is used (when *sigaction* is not supported, e. g.), it has to be taken into account that another signal of the same type must not be delivered during execution of the signal handling routine.

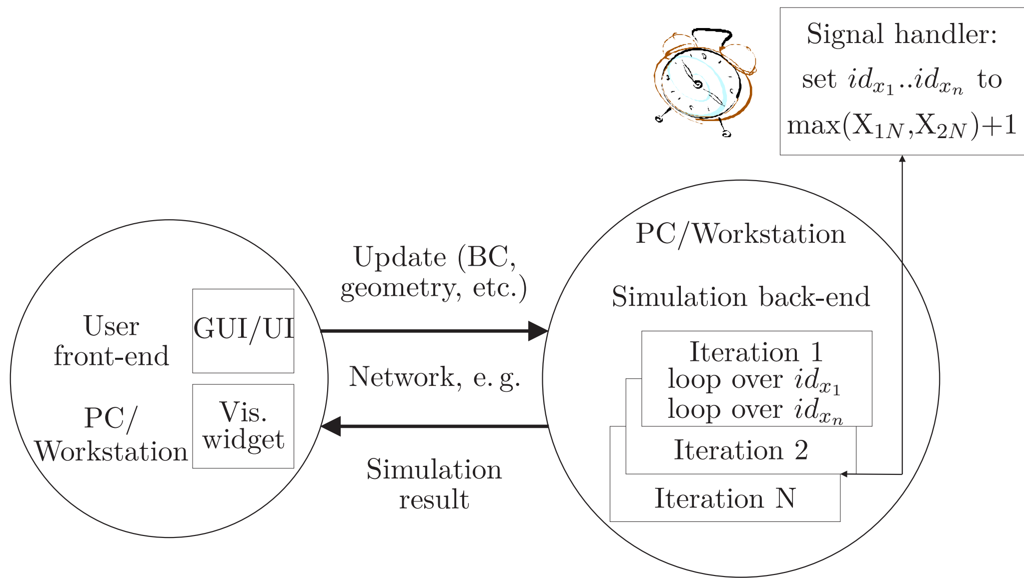


Figure 3.3: On the simulation back-end, an iterative function is running. Iterator indexes $id_{x_1}..id_{x_n}$ are reset by the SIGALARM handler to the value outside of the corresponding iterator scope, for instance max(X_{1N},X_{2N})+1. This way, the iterative function omits the rest of the calculation and starts anew.

To restart interrupted system calls, one can use SA_RESTART flag within *sigaction*. *sigprocmask* can be used in the parts of the code, where it is undesirable to receive the signal.

Any use of non-reentrant functions, such as *malloc* and *printf* inside the signal handler is also unsafe. When finding place to re-initialise the data for the new computation, it should be taken care that no new memory is allocated within the signal handler itself.

Ensuring data consistency

One of the important tasks, when one uses signals, is to ensure that certain types of objects which are being modified both in the signal handler and the main computation are updated in an uninterruptible way. Namely, such updates are said to be atomic, thus, it is impossible for the object to be in an inconsistent state during the update. However, the types that support atomic updates are usually very simple (e. g., integer). The C standard provides the specific type for this purpose. Objects of types other than sig_atomic_t *are* allowed within a signal handler under the condition that they are never accessed by the program outside of the signal handler's context. If they are, sig_atomic_t *has* to be used. Although this topic gets into operating system dependent problems, no matter how the operating system handles the issue, using this certain type for loop counters one can be sure he will not end up with corrupted bytes due to interrupts.

This is what C99 says about sig_atomic_t:

- (7.14 <signal.h>, paragraph 2) The type defined is sig_atomic_t, which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
- (7.14p5) If a signal occurs other than as the result of calling the abort or raise function, the behaviour is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as volatile sig_atomic_t.

The point of relevance for the framework itself is that handlers for SIGALRM, SIGVTALRM, SIGPROF, SIGINT, etc., which return normally, may modify some global variables. Typically, the same variables – for example, iterator indices within a loop – are being examined by the program during normal operation. For instance, when CPUs require more than one instruction to write data of a certain size to memory, a sudden interrupt can lead to data inconsistency, as explained in the following example.

A computation is performed on a machine with 32-bit long integers, 32-bit CPU registers, 64-bit long long integers. This is how the generated assembly code would look like for the C-code line:

```
x = 1; /* long x */

move #1, register
 store register, [memory location for x]
```

But if y is long long (and thus 64 bits wide), it requires two separate stores to set it to a new value – for instance:

```
 y = 0x0000000000000001LL; /* long long y; */
```

might compile to:

```
move #0x00000000, register
store reg, [memory location for y]
move #0x00000001, register
store reg, [(memory location for y) + 4]
```

Now, suppose that a portion of C code compiles into:

```
 load register, [y]
 push reg  (*)
 load register, [y+4]
 push reg
```

Suppose further that it is managed to invoke the signal handler right when the processor has finished the first *load* but has not yet started the second *load*, i. e., is about to execute (or has just executed) the instruction marked (*). If y used to hold the value 0x0000000000001LL, and it is supposed to be re-initialised with 0x0000000000000LL the code will have handled the first half – 0x00000000 – but not the second (depending if it is big- or little-endian architecture). The signal handler keeps y value intact – 0x000000000000001LL, and only when its execution finishes, the CPU will resume and execute the second *load*. This would not happen only for 64-bit variables – but also if CPU has 32-bit integers and 16-bit registers, or 16-bit integers and 8-bit registers, etc. For one of the first examples in this chapter, Code 3.3, this all means that without declaring *keep_computing* sig_atomic_t, its value might not be set to 0 when desired.

And even if the implementation happens to match particular CPU hardware – either by accident, or by knowing all the details about the CPU – what if the compiler chooses to decide to modify only the lowest byte of a variable? The keyword *volatile* is likely to inhibit such an optimisation.

Ensuring memory consistency

In the C, C++, C#, and Java programming languages, a variable or object declared as volatile usually has properties restricting compiler optimisation and/or related to threading. It is, namely, intended to prevent the compiler from performing any optimisations of the code which assume that values of variables cannot change "suddenly".

Let us consider the case of a loop which tests the same memory location repeatedly. Most of the modern compilers would arrange to reference memory only once and copy the value into a register to be used further on, speeding up the loop this way. Naturally, "this is undesirable for objects which are subject of sudden change for reasons which cannot be predicted from a study of the program itself" [40]. Thus, to ensure correct program execution, every reference to such an object has to be genuine.

This type qualifier is used within the framework implementation to:

– allow uses of variables between *setjmp* and *longjmp*,

– allow uses of *sig_atomic_t* variables in signal handlers.

The definition and applicability of the volatile keyword differs in different programming languages. These differences should be carefully considered. In any case, when using volatile, one may observe that the produced assembly code is bigger than the one without volatile specifier, because the volatile keyword stops the compiler from performing optimisations. Thus, volatile should be restricted to the variables where it is really needed. In the framework implementation, disabling this kind of compiler optimisation can be well compensated for in terms of the execution time. This holds especially if a user process and a simulation

process communicate via message passing, which is a typical scenario. The compensation for disabling some compiler optimisations is achieved by reducing the expensive communication calls for probing of an update from the user's side. Namely, these calls are now placed within the signal handler, where a user has more control over the checking intervals.

To sum up, the one guarantee from the ANSI/ISO C standard, is that one can assign to a *volatile sig_atomic_t* variable and see either the old value or the new value, never any inconsistent intermediate one. Using any other type, the C-level guarantee is gone and it has to be looked elsewhere for guarantees. If a user, based on his architecture information can make sure that no inconsistent state may occur, he can even remove sig_atomic_t and use any desired data type, e. g. integer.

Finally, Code 3.3 is a simple example of a program where sig_atomic_t and volatile must be used to guarantee correct program execution independent from a hardware architecture and a particular test case. It executes the body of the loop until it has noticed that a SIGALRM signal has arrived. When the signal arrives, the iteration in progress is allowed to complete before the loop exits.

Many applications that are amenable to concurrent execution can be programmed using either shared memory or message passing algorithms. The basics of parallel programming are presented in Chapter 2. These will be discussed with regard to signals and the interactive computing framework.

The sections to follow describe the simulation patterns, which utilise either shared memory (e. g. OpenMP/POSIX threads) or "hybrid" parallelisation models (i. e. MPI and OpenMP) in conjunction with the framework. This involves additional synchronisation issues, due to the "restart"-signals from the thread/process which is first informed about the changes, and thus, has received the updates, to the rest of the threads/processes.

Other challenges

In addition to the guaranteed data values consistency necessary for the correct program execution, a few steps have to be taken to prevent potentially introduced severe memory leaks before the new computation is started. This is due to the interrupts and their possible occurrence before the memory allocated in a simulation program – a solver of a system of equations, e. g. – has been released. Therefore, within a function which is executed asynchronously with a signal handler, an access to all the previously allocated memory has to be assured also *after* the signal handler returns.

3.1.4 Example

To make this section complete, a practical example is provided. The framework is illustrated in conjunction with a Gauss-Seidel sequential solver, implemented for 2D heat conduction simulation, where approximation is done by central

```
struct itimerval itv;
sig_atomic_t volatile id_x1, id_x2;
alarm_signal_handler()
{
  id_x1 = (Imax > Jmax)?Imax : Jmax;
  id_x2 = (Imax > Jmax)?Imax : Jmax;
  setitimer (ITIMER_VIRTUAL, &itv, NULL);
}
struct sigaction new_action;
new_action.sa_handler = handle_alarm;
sigemptyset (&new_action.sa_mask);
new_action.sa_flags = 0;
sigaction (SIGALRM, &new_action, NULL);

itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 1000;
itv.it_value.tv_sec = 0;
itv.it_value.tv_usec = 1000;
setitimer (ITIMER_VIRTUAL, &itv, NULL);

for (t = 1; t < Tmax; t++)
    for (i = 1; i < Imax; i++)
        for (j = 1; j < Jmax; j++)
            u[i][j] = 0.25 · (u[i+1][j] + u[i-1][j] + u[i][j+1] +
                             + u[i][j-1] - f[i][j]
        }
    }
}
```

Code 3.10: Gauss-Seidel sequential version to demonstrate the framework using *settimer* "instead of" *alarm.*

finite differences. The solutions discussed throughout this chapter are collected and presented within the Code 3.10. However, a reader may notice that (re)initialisation routines are missing as well as all the communication calls. The latter are discussed separately within Section 3.5.

3.2 Framework and Multithreading

For the multithreading, Lee [117] has stated that "although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism."

In modern computing articles and technical reports, there are many similar statements: "To use multicore, you really have to use multiple threads. If you know how to do it, it's not bad. But the first time you do it there are lots of ways to shoot yourself in the foot. The bugs you introduce with multithreading are so much harder to find." [79]

The challenges when using signals introduced in the previous section arose in scenarios with sequential codes. Now taking a look at the last two quotations, a reader might already suspect that the additional challenges introduced with multithreading are not negligible. To the contrary, even as far as the implementation of the framework is concerned, at least several sudden unexpected behaviours in the program have been detected (all to be addressed later). The answers to particular challenges are unfortunately beyond the information provided by the common literature sources and online manuals. Nevertheless, the final success in overcoming challenges has provided an even more complete set of framework features. Moreover, this chapter might give an interested reader a solid overview of potential error sources within any multithreaded program, especially using Unix signals.

3.2.1 Signals and Multithreading

Each process has its own signal action. Within a multithreaded application, the action of a particular signal is the same for all the threads of one process.

However, there is a way to prevent that a particular thread receives that signal. Each thread in a process has an independent signal mask, which indicates the set of signals that the thread is currently blocking. Similar to a traditional single-threaded application, where *sigprocmask* can be used to manipulate the signal mask, any thread can manipulate its own signal mask using *pthread_sigmask*. A process-directed signal may be delivered to any among the threads that does not currently have that particular signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses a random thread to which to deliver the signal. A signal may be generated (and pending) for a process as a whole (e. g., when sent using *kill*, *raise*, etc.), or for a specific thread (using *pthread_kill*).

3.2.2 The Concept of the Framework

If OpenMP/POSIX threads, for instance, are used in the initial application, as soon as a random thread is interrupted at the end of the user-specified interval (Fig. 3.4), it checks if some information regarding the user activity is available, through using the functionality of the Message Passing Interface. If the aforesaid probing of the user's message indicates that a change has been made, the receiving thread instantly obtains information about it and notifies all the other threads that their computations should be started anew.

As in the case of a single thread of execution, as soon as an interrupted thread becomes aware that there is a user-made change to be applied, it automatically

manipulates its loop variables and, thus, computation will be restarted immediately after the end of the current, innermost loop step. This is illustrated in Algorithm 5. Our implementation, favourably, issues clean termination of OpenMP parallel loop and clean exit from the parallel region after the threads are either implicitly or explicitly synchronised and before the whole iteration is resumed. In this manner, the correct program execution will be ensured.

Algorithm 5 Idea of the framework – multithreaded scenario

```
 1: X_1N, X_2N declared global, sig_atomic_t, volatile
 2: alarm_signal_handler():
 3: X_1N, X_2N ← −1
 4: reset_alarm()
 5:
 6: //Inside the main function:
 7: init_sigaction()
 8: set_alarm()
 9: for (t ← T_0 to T_N) // iterations over time do
10:    parallel region private(idx_1, idx_2)
11:    for (idx_1 ← X_10 to X_1N) do
12:       for (idx_2 ← X_20 to X_2N ) do
13:          // can be interrupted at any point, X_1N, X_2N are
14:          //reset in the interrupt handler
15:          process(data[id_x1][id_x2])
16:       end for
17:    end for
18: end for
```

3.2.3 Challenges

The challenges for integrating the framework in multithreaded scenarios are, as expected, more numerous than for the sequential scenario. They refer to visibility of variables private for any thread but the main one, i. e. the one with thread identification 0, as well as the usage of critical sections, mutexes, semaphores, etc. within the initial code.

"Jumping" out of a parallel region

According to OpenMP standard, it is not allowed to "jump" (using *sigjmp* mechanism) out of the parallel region, and then back, thus, it is not desirable to keep the initial *sigjmp* concept of the framework also for the multithreaded scenario.

Visibility of the variables in the signal handler

For the second approach – manipulating the loop indexes – an issue which occurs for OpenMP, for example, is that private loop variables are not visible within the signal handler. Any attempt to manipulate id_{x_1}, id_{x_2} would fail. Thus, as a

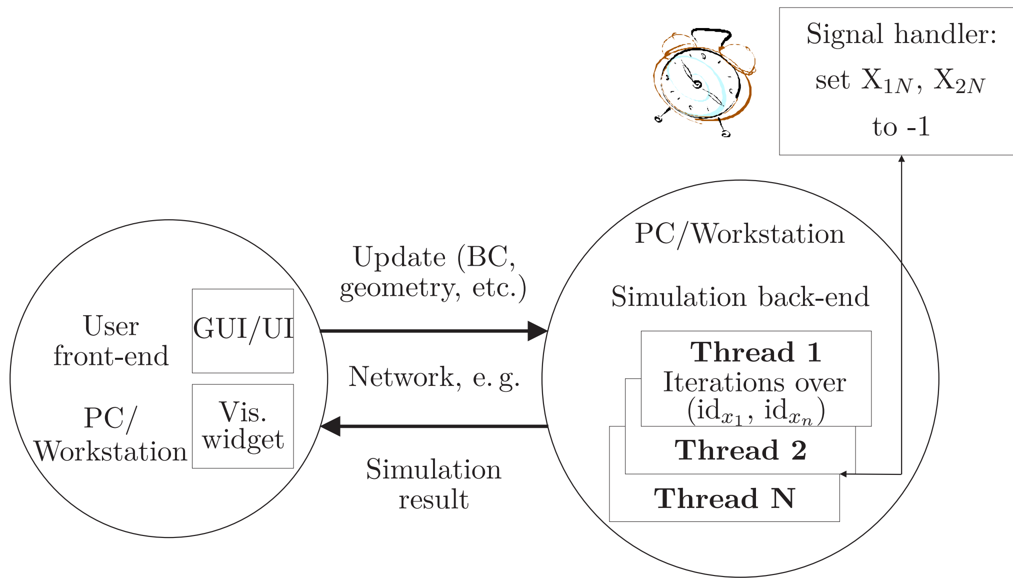


Figure 3.4: The framework with manipulating loop indexes; Notation id_{x_1}, id_{x_2}, X_{1N}, X_{2N} refers to the variables from Algorithm 5

reader may notice in Fig. 3.4 and Algorithm 5, the concept used for multithreaded simulation is based on manipulating the global variables I_{max}, J_{max}.

Critical sections and mutexes

All the mutual exclusion (mutex), semaphores, or any kind of locking-based variables have to be declared globally visible, so that they can be unlocked in the signal handler, and the return to the main code is possible (as illustrated in the Code 3.11).

In general, there seems to be less "control" concerning this matter in OpenMP than, for instance, in POSIX threads, since in OpenMP many parallel constructs have implicit locking effects. Branching out of a critical section, i. e. using breaks, returns, exceptions is not allowed by OpenMP standard. A user still has a few options. He can modify his code, i. e. completely block SIGALRM in those particular parts of an OpenMP-parallelised code. Otherwise, despite of some advantages of critical sections over scoped locking (e. g., it is typically a faster mechanism), according to the framework requirements, critical sections should be replaced by mechanisms for handling mutual exclusions, such as locks. The corresponding variables should be globally visible, thus, it should be possible to unlock them within the signal handler (Code 3.11). Only the thread which owns the lock at the point when an interrupt happens can also release it (unlock) and, hence, if any other thread catches the signal, the unlocking does not have any effect.

```
omp_lock_t lock;

void handle_alarm (int)
{
  omp_lock_unset (&lock);
}

// Within the main function:
omp_init_lock(&lock);
#pragma omp parallel
{
  omp_lock_set (&lock);
  do_smt();
  if(omp_lock_test (&lock))
    omp_lock_unset (&lock);
}
```

Code 3.11: Signals and locks

3.2.4 Example

An example pseudo-code (see Code 3.12) of Jacobi solver is provided (for 2D heat conduction simulation, approximated using finite differences) – this time running multithreaded. Specifically in OpenMP, the signal-handler manipulation of the upper limit for the index vector for which the parallelisation is done, would still not have any effect on the loop execution. With other threading libraries, this issue does not occur, since the developer has more control over the visibility of variables (at the expense of having more responsibility).

A reader may observe that the communication scheme with a user process is again omitted in the provided example code, since it is discussed in more detail within Section 3.5, however, for a brief overview at this point, a reader may refer to Fig. 3.4.

3.3 Framework and Distributed/Hybrid Scenario

When computing environments supporting both parallel paradigms mentioned at the beginning of this subsection are available, a hybrid algorithm, at a sufficiently high level of granularity, may be used to advantage [158]. Computing environments which possess the hardware diversity required for such parallel applications nowadays, i. e. provide a pillar for multiple concurrent computation models, are the norm rather than the exception.

```
alarm_signal_handler()
{
  Imax = −1; // Does not have any effect on the loop only for OpenMP.
             // Still, since all the inner loops are skipped,
             // the overall effect remains similar also in this case.
  Jmax = −1;
  ualarm (1000); // time in microseconds
}

// Within the main function:
struct sigaction new_action;
new_action.sa_handler = handle_alarm;
sigemptyset (&new_action.sa_mask);
new_action.sa_flags = 0;
sigaction (SIGALRM, &new_action, NULL);

ualarm(1000);

for (t = 1; t < Tmax; t++) {
    reinitialise_u_old();
    // here start a parallel region, private(j)
    for (i = 1; i < Imax; i++) {
        for (j = 1; j < Jmax; j++) {
            u[i][j] = 0.25· (u_old[i + 1][j] + u_old[i − 1, j] + u_old[i][j + 1]
                            +u_old[i][j − 1] − f[i][j]);
        }
    }
}
```

Code 3.12: Jacobi multithreaded for Poisson equation code segments, demonstrating the integration of the framework.

3.3.1 Signals and Distributed scenario

The major subject in the implementation of the framework for this scenario, where the simulation processes communicate over message passing protocols, is that all the "signals", i. e. user updates, have to be explicitly "transferred" and applied for each process. A challenge is to broadcast a "signal" efficiently to all the processes. A broadcast algorithm had to be implemented within the framework to be used instead of MPI_Bcast, because there was still no probing of a message possible for MPI collectives. With MPI-3 standard, however, comes a non-blocking Bcast implementation, such that an arrival of a broadcasted message can be tested for. Thus, it might be possible soon to use vendor-optimised implementation as well and re-implement the framework's Bcast.

3.3.2 The Concept of the Framework

In the case of a pure distributed scenario, each process has its own alarm which rings (Fig. 3.5). In the case of "hybrid" parallelisation of a simulation, very similar to this, a random thread in each active process is being interrupted, hence, fetches an opportunity to check for the updates.

In both cases, if no user action has taken place, the previous computation is continued (for the "hybrid" scenario – without stopping the other threads). Otherwise, the user data is received, the "restart signals" are sent to other processes and, thus, the loop counters are manipulated in all the required computations.

3.3.3 Challenges

From the framework point of view, the difference in comparison to the multithreaded parallelisation is that now all the processes have to be notified about the user action, which involves additional message passing overheads. In the current state of development, it is assumed that one master process, which is the direct interface of the user's process to the computing-nodes (i. e. slaves), apprises all the slave processes of the user-made alterations, which, in the case of demand for and availability of a large number of slaves may result in the master process becoming a bottleneck.

Synchronisation of the processes

The processes might need to synchronise at the end of each iteration to exchange, e. g. ghost-cells' data. Then, they may proceed. But the question which remains is how the "alarms" are synchronised for each of the processes, how fast all of them can be notified, and what happens in the case of perpetual user interaction. What happens if, as it is currently implemented in a test scenario, an update occurs approximately each 5 milliseconds?

Two non-blocking broadcast algorithms have been developed in order to compare the performance. It should be noted that the implementations have been finished before MPI-3 standard has introduced non-blocking collectives. However, at the

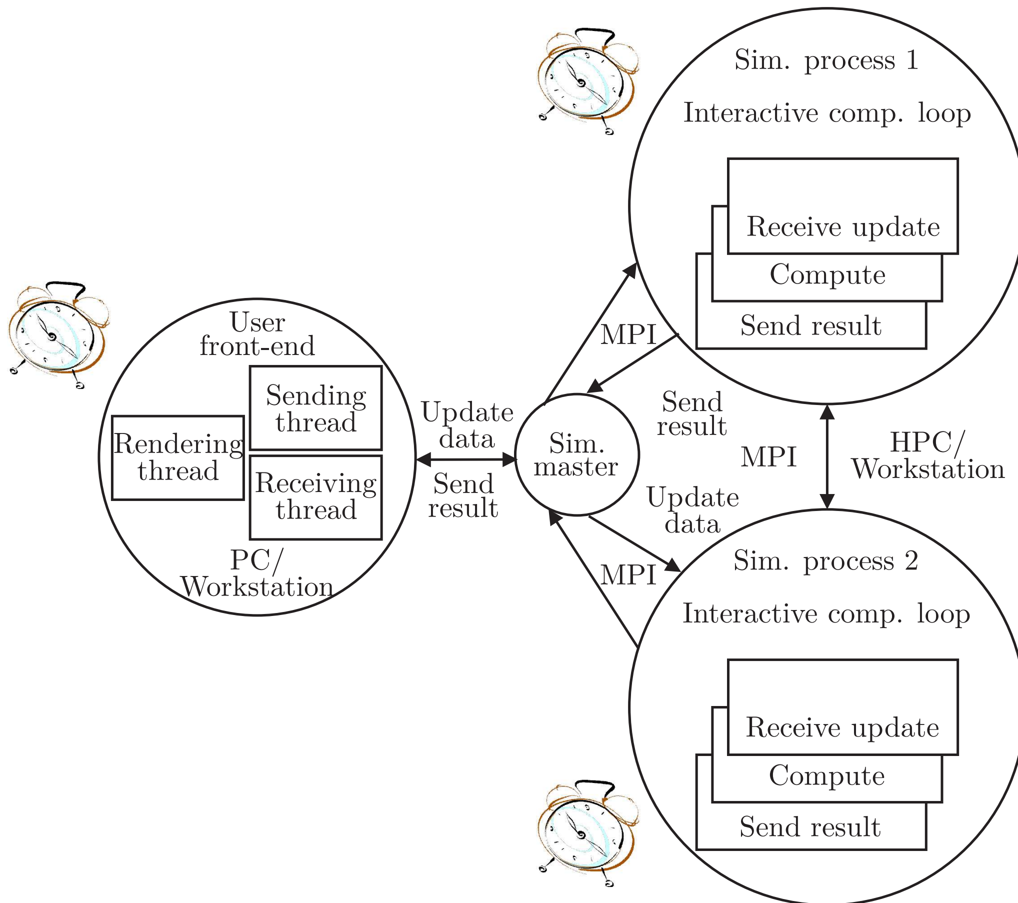


Figure 3.5: Framework – distributed scenario

time this work has been written, it is all still at an early stage and not all the implementations are yet conforming to the new standard. Consistency protocols are used to verify that all the tasks in this distributed scenario apply the user changes consistently with one another.

Linear broadcast algorithm

The first distributed test-environment has three components. First is the described simulation back end, which consists of n slave processes executing Jacobi solver instructions in parallel, as previously described. Now with the framework integrated, each of the processes has its own alarm set to “ring” each millisecond. Second is the user process, sending updates of the boundary conditions – in our example the temperature – each 5 milliseconds. Third is a framework entity – a master process, which is actually the first one receiving the user updates, forwarding them to all the simulation processes and collecting results. It has its own alarm set to 2 milliseconds. A simple linear broadcast algorithm has been implemented – the master process forwards the updates to all the slaves, one after the other (Fig. 3.6).

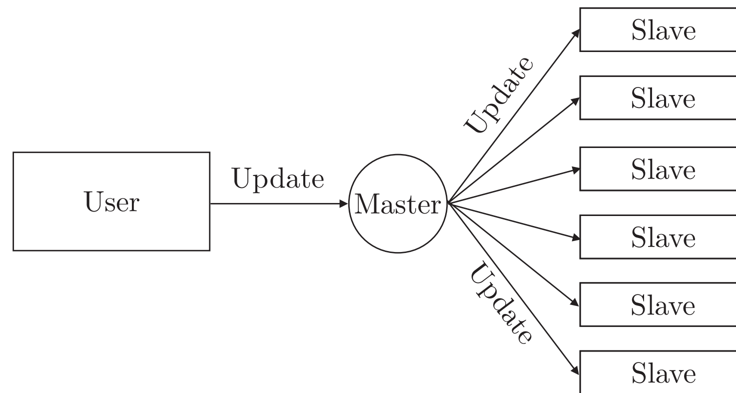


Figure 3.6: Linear broadcast

Hierarchical broadcast algorithm

To avoid potential bottlenecks of one master process in a massively parallel scenario, another version of efficient broadcast algorithm for transferring the signal to all computing nodes has to be developed (Fig. 3.7). Moreover, there is a trade-off between ensuring a minimal number of checks per process and receiving the update data promptly. In the case of a larger number of processes, where the difference might be distinguishable, it is worth experimenting with different alarm intervals on different levels of the hierarchy in order to reach the optimum. The results in terms of comparison of the time needed to pass an update message

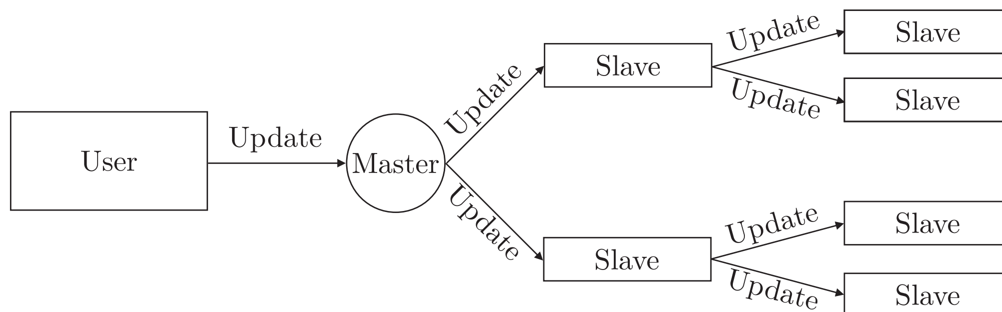


Figure 3.7: Hierarchical broadcast

to all the active processes for these two broadcast algorithms are provided in Subsection 3.7.2.

3.3.4 Example

A simple distributed parallel MPI-based Jacobi solver is implemented based again on Finite Difference approximation of 2D heat conduction simulation. The data (in terms of temperature) is read in at the beginning of the simulation program, so that each process can start his own computations. At the end of the iteration, the processes synchronise and exchange data as shown in the Fig. 3.8. The result

is afterward sent to a user. This is done mostly only for testing purposes – to see how fast the signals are "passed" from the user process to all the other processes. For a more computationally efficient algorithm, a reader is referred to Subsection 2.3.7.

The user process, on the other hand, is simply generating some update data and sending it to the simulation in very small intervals, i. e. 5 milliseconds (which is very unlikely in real applications). The results in terms of the overhead of the framework, compared to the execution time of the initial simulation, are based on this particular example and presented in 3.7.1.

3.4 Suggested Hierarchical Approaches

Although CPU power and memory capacity of computers have increased in the last decades, many numerical problems are still only tractable with simplifications and accuracy trade-offs. Moreover, it is necessary to estimate the quality of the applied approximation, before one can rely on the result while making decisions in "real-life" situations.

Adaptive procedures try to automatically and optimally adjust the model to achieve a solution with a needed accuracy. These adjustments may, e. g., refer to refining or coarsening of a mesh, and/or adjusting the basis functions used within different approximation methods.

Typically, a trial solution, with lower accuracy, is generated first, i. e. on a coarse mesh, or with a low-order basis. Then, the error of this solution is estimated, to check if it satisfies the prescribed accuracy. Next, if needed, minimal-effort adjustments are made to gain the desired solution. For example, a discretisation error might be reduced to its desired level using the fewest degrees of freedom, i. e. deformation state variables for the nodes (in the Finite Element approximation, e. g.). A numerical solution can be improved by, e. g., refining the grid (i. e., increasing the number of mesh cells), increasing the local expansion order (i. e. polynomial order), etc. Here are listed some of the common refinement methods:

1. h-refinement (increasing the number of mesh cells),
2. p-refinement (switching to higher order polynomials as shape functions),
3. hp-refinement (combination of h and p refinements),
4. r-refinement (adjust the positions of the nodes),
5. m-refinement: one switches to a different equation, i. e. physical model depending on the local behaviour of the approximated solution. As an example one may use linearised equations only if the nonlinear terms of the physical model are negligible.

In addition to the previous taxonomy, refinements can either be:

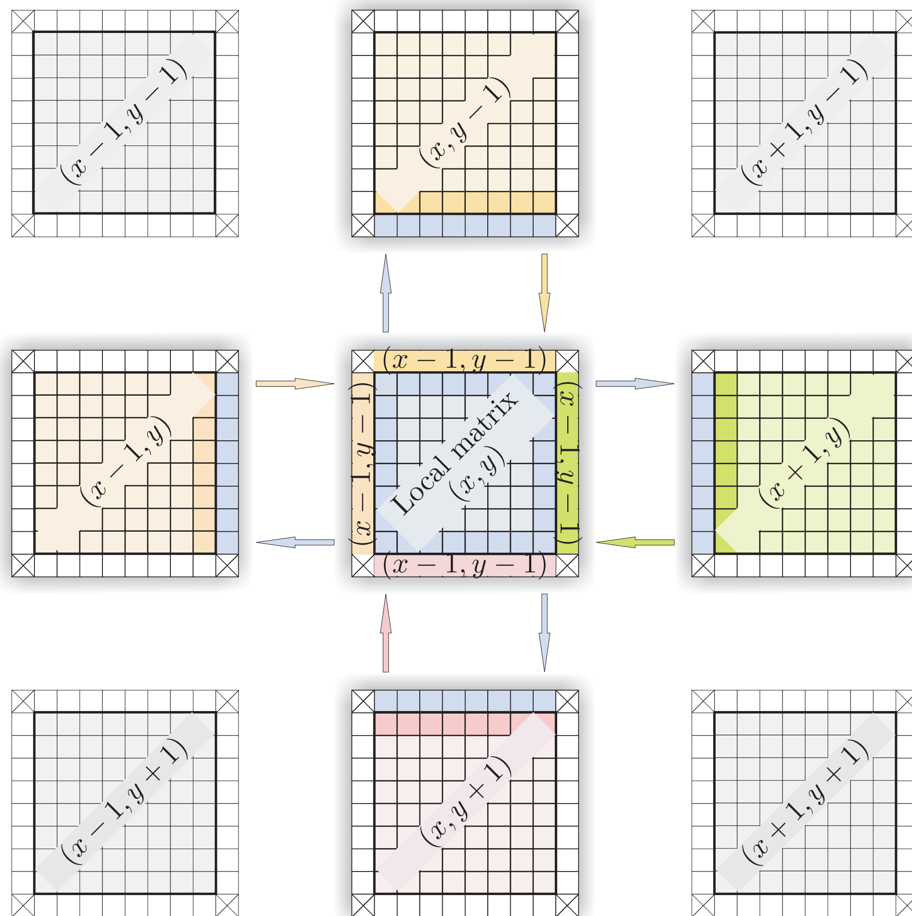


Figure 3.8: Distributed Jacobi algorithm: For each $(n \times n)$ matrix assigned to one processor, it keeps $((n+2) \times (n+2))$, where 2 vectors $(n \times 1)$ and 2 vectors $(1 \times n)$ are so-called "ghost layers", i. e. store the data received from the "neighbour" processes. The received data is the needed part of the result of the previous iteration executed by the "neighbour" processes, which is necessary for an update in the current one.

1. uniform, i. e. when all mesh cells are divided, or polynomial order is uniformly increased;
2. selective, where only chosen mesh cells, in the regions of interest, will be divided/will have their polynomial order increased.

A user can profit most from the framework if he uses hierarchical simulation approaches in conjunction with it. In the case when an interrupt occurs, the computation using coarsest/lowest refinement level can be started, switching gradually to finer resolutions/polynomial orders, e. g., only when a user stops interacting. Due to the unpredictability of the underlying simulation model and simulation methods, the support for hierarchical approaches is neither a part, nor responsibility of the framework.

However, some kind of a hierarchical approach *is* addressed in addition to the basic concept of the framework in most of the application scenarios, as described in Chapter 4. The author considers further possibilities, depending on a particular application scenario, inexhaustible. In future test applications, the same or similar ideas might be re-used, or their range extended.

Therefore, a very brief overview of most common adaptive scenarios is provided, with a remark that the extension of the set of possibilities, altogether with implementation details, can be found in Chapter 4. In the same chapter, the corresponding performance results are provided, as well as more concrete conclusions, especially concerning the effectiveness of the framework in combination with adaptive approaches. In this chapter, however, only a brief overview of potential approaches is considered, and the possibilities in conjunction with the framework discussed rather on an abstract level (with few exceptions). First, h-refinement is discussed, where the polynomial order is kept fixed and selective subdivision of the mesh is done.

3.4.1 H-refinement

Adaptive grid generation assumes establishing an initial discretisation of a given domain, then refining a given discretisation when needed, and adjusting it.

Initial discretisation starts from a description of the domain, which is then being decomposed into simplicies or quadrilaterals, e. g., conforming to some ‘regularity’ criteria. Given a vector of local error indicator, a new discretisation providing smaller local error should be achieved. Elements which should be refined are marked, replaced by some with predefined smaller diameters and, thus, a new grid is constructed. The main challenge is to avoid potential degeneration of the simplices during repeated refinement and the “matching” between the coarse and the fine zones, and often also improve boundary approximation. These steps can be repeated as long as the grid is still too coarse to satisfy the imposed criteria.

Adaptive mesh refinement can be described best within the Finite Element context. Namely, the aim is to use as less as possible degrees of freedom, however,

the approximated solution must have a required accuracy. The usual FEM analysis would proceed from the selection of a mesh. To determine whether or not the mesh is optimally refined, a solution on a finer mesh is typically calculated and a comparison of the two solutions is made. If needed, the existing elements could either be divided into smaller ones, i. e. by adding more nodes, keeping the original element boundaries intact, or a complete re-meshing has to take place. Then, a new set of element shape functions is developed for all the nodes. The shape functions may be generated from the Lagrange polynomial [182].

An obvious disadvantage of the higher order h-elements for introducing hierarchical approaches within interactive computing environment is that "a new set of shape functions is required upon changing the element order" [143]. All calculations of element stiffness matrices and element load vectors have to be repeated. Therefore, this adaptive technique is not very favourable from the interactive computing point of view, since it does not offer straightforward possibilities to profit from the previous calculation, i. e. use a result on a coarser grid for the computation on a finer one. However, a simulation can keep doing "faster" computation for smaller h and then, once a user stops interacting, start from scratch, calculating for higher h.

A converging adaptive algorithm for linear elements applied to Poisson's equation in 2D, e. g. can be found in [68]. An error is estimated starting from a coarser triangulation, and a sequence of refined triangulations and approximate solutions are generated. Another example of a method to generate a sequence of successively finer or grids on enlarged domains until the desired accuracy of the solution is reached can be found in [42].

3.4.2 Up- and Down-Sampling

Due to the mentioned disadvantages for h-refinement within classical h-FEM, alternatives should be considered. For some problems, especially those which have smooth solutions away from the boundaries, when doing h-refinements on a structured grid, e. g., it is possible to do simple upsampling from the coarse grid to the fine grid and use it as an initial guess for the solution on a finer one.

In the 2D heat conduction simulation in Chapter 4, upsampling has resulted in (visually) satisfying solutions. In applications such as tone mapping, bilateral upsampling [112] from a coarse grid solution produced also good results. In [157] global progressive upsampling of the edited image is done. To improve solution at the seams, local refinement has been performed. Then, an iterative method (CG or SOR) runs on original pixel data with higher resolution.

Downsampling can be used when switching again to the coarser grid, e. g. in the case of repeated user interaction, where some kind of averaging of the values on a finer grid can be used to get an initial guess value on a coarser grid. Both of these techniques would be applicable to the AGENT test scenario in Chapter 4. They would typically involve intergrid transition functions, similar to those used in multigrid algorithms, as illustrated in Eq. 2.12 and 2.13, for instance.

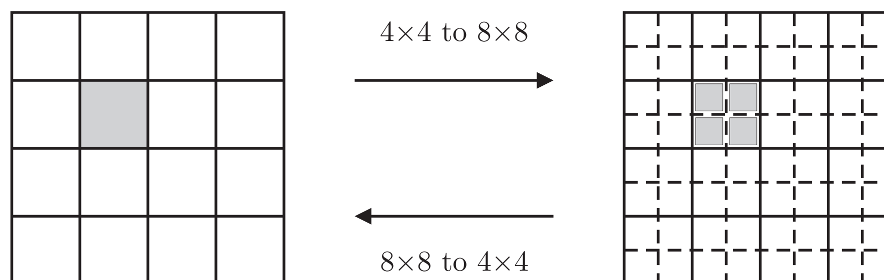


Figure 3.9: Upsampling (a value from a coarser grid is assigned to the corresponding grid cells of a finer grid) and downsampling procedure (by averaging values on a finer grid, the corresponding value on a coarser grid is gained).

3.4.3 P-refinement

In the *p*-refinement methods [145, 144, 73, 131, 19, 159], the initial mesh elements are kept unchanged, and selective or total increase in the polynomial order is applied. However, the polynomial order, *p*, of the shape functions can be enriched by adding higher order term(s) without changing the existing lower order shape functions. These shape functions are generated, for example, from Legendre polynomial. They are called *hierarchical shape functions* [182].

Adaptive *p*-refinement is conceptually simpler than adaptive *h*-refinement [160]. The hierarchical shape functions allow for the re-usability of all existing element stiffness matrix terms in the next solution. As a result, *p*-refinement seems to be a better adaptive refinement approach [145, 144, 73, 131, 19, 159, 143].

If uniformly increasing *p* for all elements in FEM, the order of the shape function in the mesh is uniformly increased until the convergence is achieved according to the desired accuracy. In many cases, extrapolation type error estimators are used [19].

In non-uniform *p*-refinement, the order of shape functions is increased in some selected elements, if an error is large. For this, an error indicator is required, i. e. a feedback parameter calculated from the available information of the previous solution, which is used to select the region or the elements for refinement [143]. The use of non-uniform *p* analysis in adaptive *p*-refinement will result in an improvement in computational time for solving the equations and less memory usage. The issues which could arise related to having to store many records to track all the information and data needed for mesh upgrading are discussed in [143]. Namely, element sub-stiffness matrix terms which correspond to the interaction between the degrees of freedom corresponding to the high order shape functions and the existing degrees of freedom must be stored in the places where an equation solver and an error estimate can use them [143]. Once this is provided, interactive computing framework could be easily plugged in to help switch to different *p*-refinement levels when appropriate. This means to switch to a lower

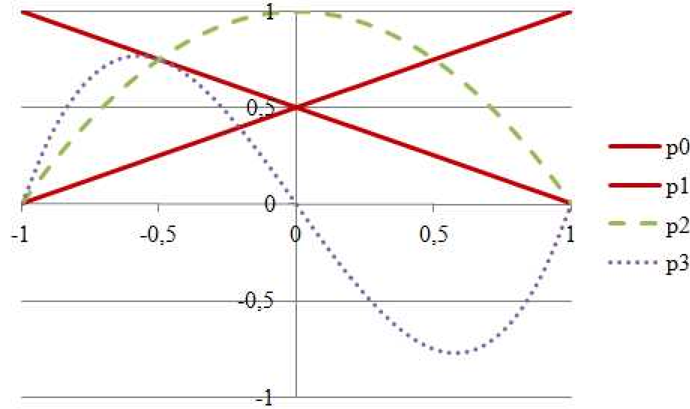


Figure 3.10: Hierarchical shape functions for $p = 1$ (p_0, p_1), $p = 2$ (p_2) and $p = 3$ (p_3).

one when there is an update, then gradually to a higher one and fetch all the intermediate results to provide feedback to the user.

There are polynomial sets, such as Legendre polynomials, which retain an orthogonal property with the increase of polynomial order [182]. It allows the polynomial order to be increased without changing the existing lower order ones, i. e. the element higher order shape functions are simply added to the existing shape functions for each desired element. For example, in 1D, a "bar" element has its natural coordinate ξ in the range $[-1, +1]$ and its hierarchical shape functions are given in Eq. 3.1 [182] (see also Fig. 3.10).

$$\begin{aligned} p_0(\xi) &= \tfrac{1}{2}(1-\xi) \\ p_1(\xi) &= \tfrac{1}{2}(1+\xi) \\ p_p(\xi) &= \tfrac{1}{(p-2)!2^p}\tfrac{d^{p-2}}{d\xi^{p-2}}[(1-\xi^2)^{p-1}], \quad p = 2, 3 \ldots \end{aligned} \tag{3.1}$$

After rearrangement, the corresponding finite element equation can be written as

$$\begin{bmatrix} k_{ii} & k_{ih} \\ {k_{ih}}^T & k_{hh} \end{bmatrix} \cdot \begin{bmatrix} d_i \\ d_h \end{bmatrix} = \begin{bmatrix} f_i \\ f_h \end{bmatrix} \tag{3.2}$$

and $[k_{ih}]$, $[k_{hh}]$, and $[f_h]$ are the element sub-stiffness matrices and the element load vector corresponding to the newly introduced hierarchical shape functions. An advantage of using p-elements is that the elements can be upgraded efficiently to any desired order. Element sub-matrix $[k_{ii}]$, which is calculated for the initial set of shape functions is reused in Eq. 3.2. Thus, only the $[k_{ih}]$ and $[k_{hh}]$ terms are calculated.

3.5 Data Transfer

Experience gained from the application scenarios has shown that it is essential for a user to be able to send desired updates asynchronously from receiving the previous result. The component related to transfer of a simulation result plays a remarkable role in an interactive steering process. Fast data transfer remains a challenging task for developers in the case of high accuracy requirements and, thus, very large data sets generated on the simulation side, despite of the networks with ever-growing bandwidth. Therefore, this chapter also discusses non-blocking, mostly asynchronous transfer of the update data to the simulation, as well as how one can achieve immediate transfer of a simulation result by selecting the information really needed by a user.

3.5.1 Our Framework Functionality for Communication

The communication calls used in the framework are wrappers around MPI communication calls. For sending, non-blocking routines are used, due to the requirement not to keep the sending process pending more than necessary. However, it has to be guaranteed at least that the data is copied to the send buffer and, thus, MPI_Bsend is utilised. For receiving an update, the simulation process uses a blocking receive, but only after the message is confirmed to have arrived by non-blocking and asynchronous MPI_Iprobe, which returns immediately. The simulation result is sent by a standard send function, which is blocking when a message is larger than a send buffer and non-blocking otherwise (MPI decides internally). The same result is received by a user process by a blocking receive, since this thread of execution is asynchronous with others (i. e., the thread fetching and sending user updates and the rendering one). This is detailed in the subsections that follow.

3.5.2 Fast Data Transfer

As Chapter 2 states, the improvements in network bandwidth are not the only prerequisite for fast transfer of a simulation result. They are often mandatory in interactive environments where update rates have to be extremely high, i. e. less than a second. Another prerequisite, however, are network strategies, either software or hardware based. These strategies and the state-of-the-art implementations are named in the same chapter.

In addition to the flexible asynchronous communication pattern, the framework offers a feature similar to the sliding window described in Chapter 2. Namely, a user can pre-define the sliding window size in each spatial direction and choose the left upper corner of the data he would like to preview, thus, receive only the result within this spatial frame (Fig. 3.12). In the current version, the window size is assumed to be fixed and corresponding to the visualisation resolution. The future work in this direction should cover the case when the specified size exceeds the size of the pre-defined window, when some kind of extrapolation of the values has to be done to fit the data into the given window. This way the amount of

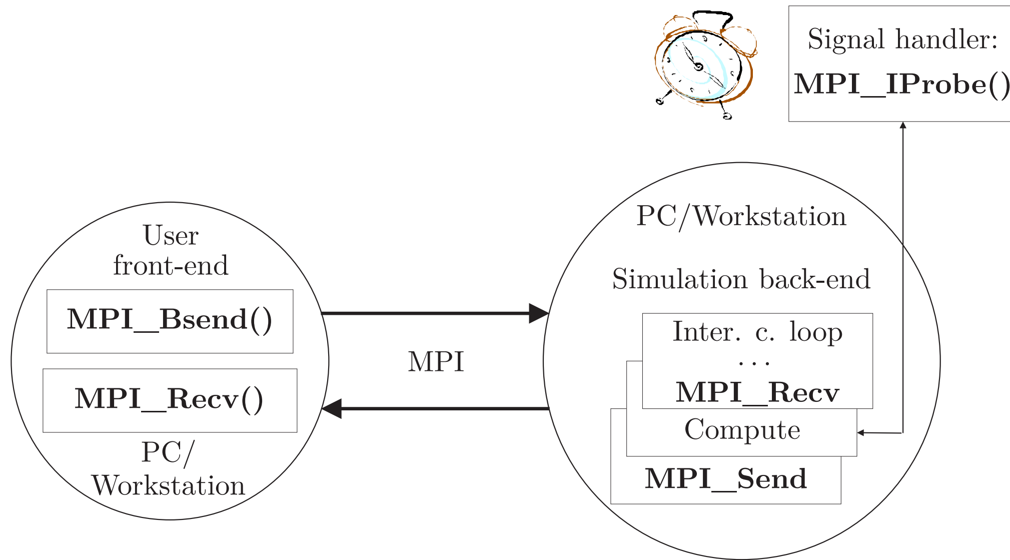


Figure 3.11: Our framework functionality for communication.

data to be transferred would remain constant all the time and restricted to a certain region of interest.

3.6 Visualisation and GUI Front-End

The experience based on the four main test cases has shown that it is crucial to keep the three separate threads – the one in charge of user interaction, the rendering thread and the communication thread – asynchronous to each other (Fig. 3.13). Otherwise the operating system could assign computational resources to the more loaded rendering thread, thus, would not allow for user interaction to take place.

Typically, it is a users' responsibility to interpret the results, since the framework cannot predict all the requirements. It is only in charge of storing the data in both sending (i. e. on the simulation side) and receiving (i. e. visualisation) buffer.

3.6.1 Challenges

When it comes to user interface and visualisation APIs, sometimes the events which should be delivered to a widget must be put into a special thread-safe queue for delivery. If for instance a rendering thread is not the main thread, different visualisation libraries provide different thread-safe objects. Qt version 3 [17] provides one such (see Code 3.13).

Similar is supported by WxWidgets library [18] (see Code 3.14).

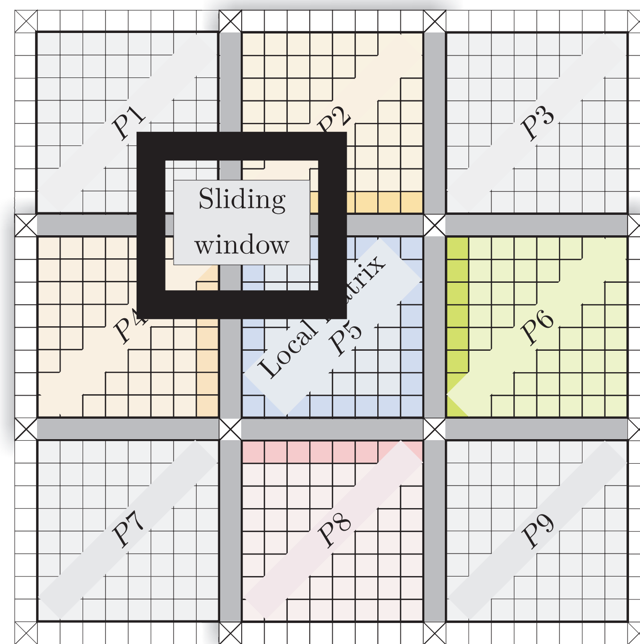


Figure 3.12: Selection of the data of interest using the concept of sliding window.

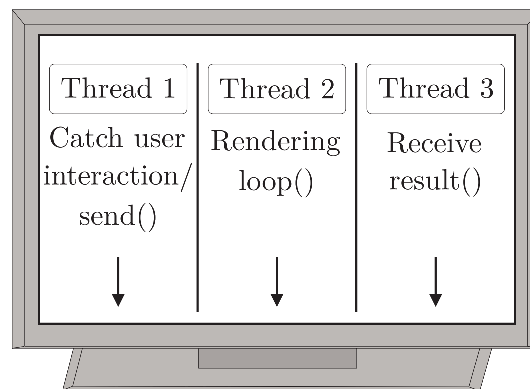


Figure 3.13: Visualisation and GUI front-end

```
// Within the rendering loop:
QCustomEvent *event = new QCustomEvent (event_id);

QApplication :: postEvent (receiver, event);
msleep (100);
```

Code 3.13: Thread safe queue of events in Qt3.

```
wxPaintEvent event (wxEVT_PAINT);
wxPostEvent (m_frame, event);
```

Code 3.14: Thread-safe queue of events in wxWidgets.

3.7 Results

3.7.1 Overhead of the Framework

The comparison of the overheads of the framework are given both for multi-threaded and distributed parallel scenario. The overhead is not significant, despite of the perpetual, non-realistic, user interaction (in few milliseconds intervals).

Results for (multi)threaded Jacobi – overhead of the framework

First, one would like to evaluate potential overheads caused by integration of the framework for a previously given multithreaded example scenario.

For the cases of single-, two- and four-threaded simulation and the different initial problem sizes: 1000×1000 (Fig. 3.14), 500×500 (Fig. 3.16), and 300×300 (Fig. 3.17), the overhead in terms of the execution time is up to 10 %. It is caused only by the cyclic signal invocation every millisecond. It is assumed that there is no update available at this time and that the simulation may immediately proceed. For the problem sizes 1000×1000 and 300×300, almost perfect (linear) speedup has been achieved, with or without integrated framework. Around 10 % lower speedup with the framework integrated, for the 500×500 scenario, is due to synchronisation issues with threads and *alarm*. Namely, it is required that the last set alarm within one iteration expires before the new iteration starts. One can improve this by finding and setting a more optimal alarm interval. Furthermore, the total execution time (and the speedup rate) in the case of an update, depends on the time when the update occurs, due to these safety-grounded synchronisation issues of the alarm and the computation. The speed of getting the result can be accelerated by a hierarchical approach. As shown in Fig 3.15, the overall execution time remains similar to the original simulation scenario, i. e. without the framework integrated, by employing this approach. The approach is based on switching from one mesh resolution to another according to the frequency of user interaction.

Distributed Jacobi, results – overhead of the framework

In none of the tests with the distributed test case, i. e. for different *alarm* intervals, has the integration of the framework significantly affected the overall execution time. This holds also when the user interaction has been invoked in 5-millisecond intervals, which is far more frequent than typically occurs in practice. The results of the measurements are shown in Fig. 3.18, where it can be observed that speedup curves remain almost intact compared to the original simulation.

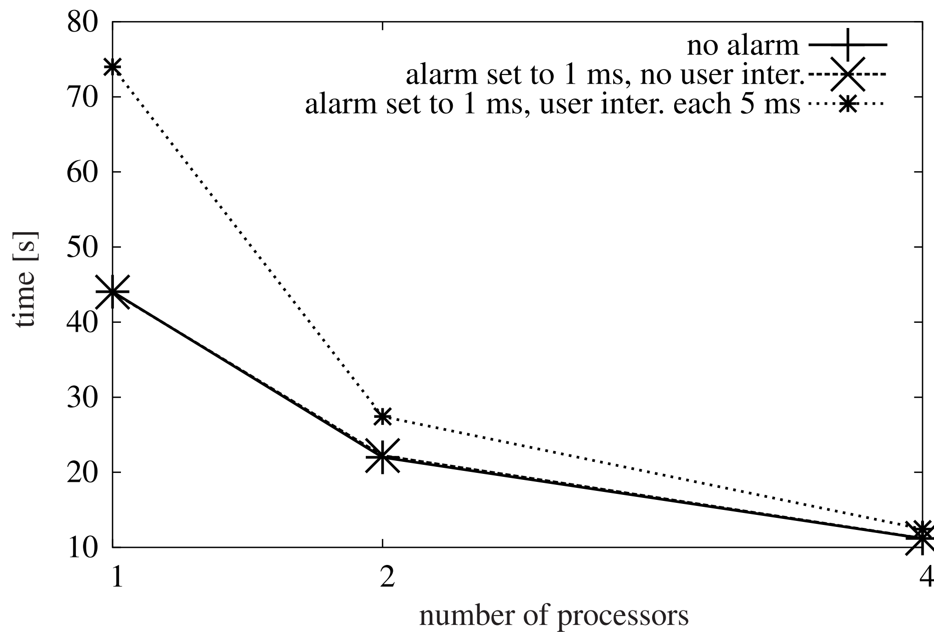


Figure 3.14: Multithreaded Jacobi solver, on 1000 × 1000 grid, user interaction rate is 5 ms. Execution times in seconds (y-axis) are provided – based on the average of several runs – for the computations on 1, 2 and 4 cores (x-axis).

3.7.2 Comparison of the Results for Linear and Hierarchical Broadcast

Measurements of the "speed" of the signal transfer to each processor – both for hierarchical and linear algorithm – are presented in Fig. 3.19. Obviously, linear algorithm performs slightly better for this relatively small number of processors. However, for a larger number of processes it might be more advantageous to use hierarchical broadcast. The result could be sent back the same way to the master process, which, finally, collects all the parts of the result. This kind of communication scheme is more advantageous when dealing with larger data.

3.8 Conclusions

In this chapter, the implementation ideas of the framework as well as some technical details are described. Since the concept of signals is used, it is discussed in the context of different codes (sequential or parallel). The implementation details are provided for different operating systems (Unix or Windows) and compilers. In addition to the challenge of combining signals with multithreading, which is not said much about in particular standards (OpenMP, e. g.), thus, involves special concern and treatment, other challenges come from ensuring memory and data consistency during sudden interrupts. Asynchronous but reliable communication patterns between the user front-end and simulation back-end and fast

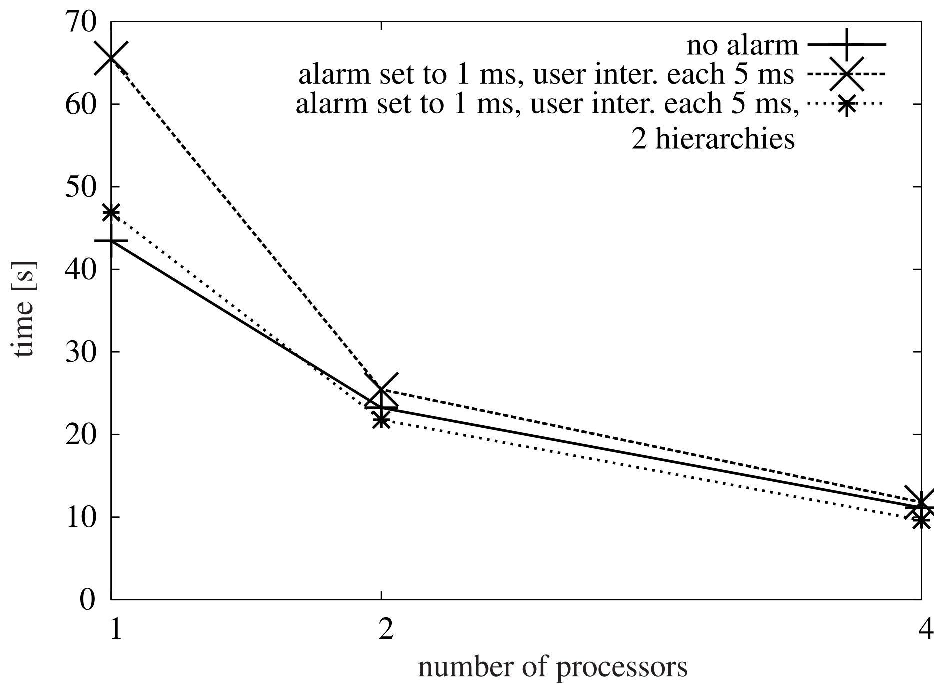


Figure 3.15: Multithreaded Jacobi solver, on 1000 × 1000 grid. 2 hierarchies are used, especially to compare the execution times with the calculation without the framework integrated (for the same number of iterations). The execution times in seconds (y-axis) – based on the average of several runs – are close to each other for the computations on 1, 2 and 4 cores (x-axis), on the expense of the accuracy for the iterations where user interaction has occurred.

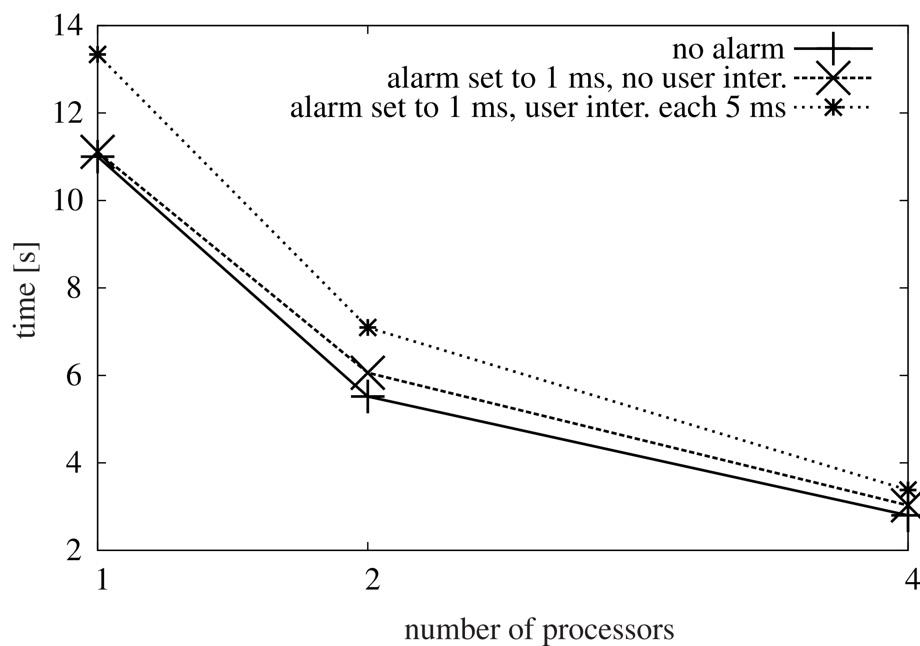


Figure 3.16: Multithreaded Jacobi solver on 500 × 500 grid: execution time measurements in seconds (y-axis) – based on the average of several runs – for the computations on 1, 2 and 4 cores (x-axis). The results are provided both for the initial program and the one with integrated framework with or without user interactions actually occurring.

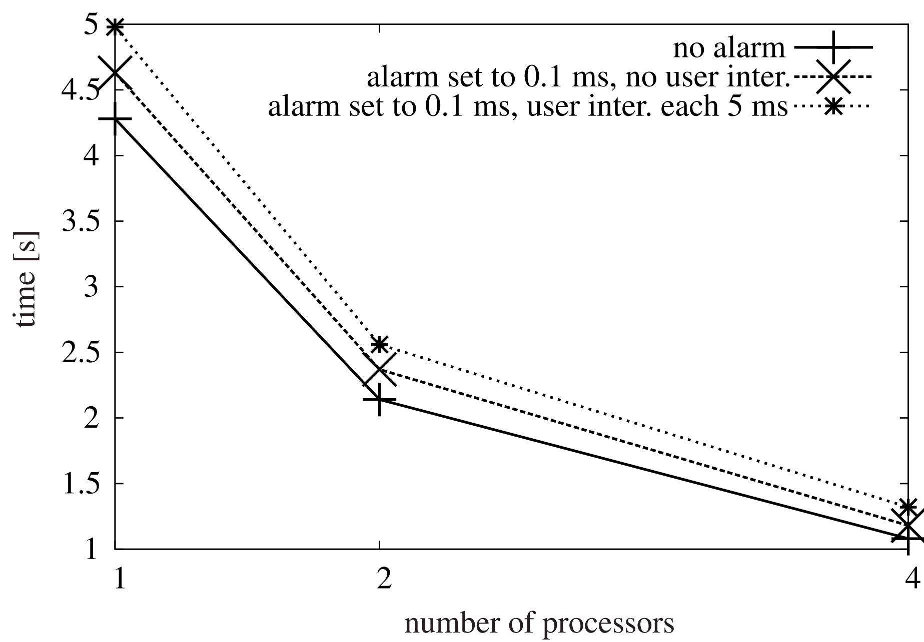


Figure 3.17: Multithreaded Jacobi solver, on 300 × 300 grid: execution time measurements in seconds (y-axis) – based on the average of several runs – for the computations on 1, 2 and 4 cores (x-axis). The results are provided both for the initial program and the one with integrated framework with or without user interactions actually occurring.

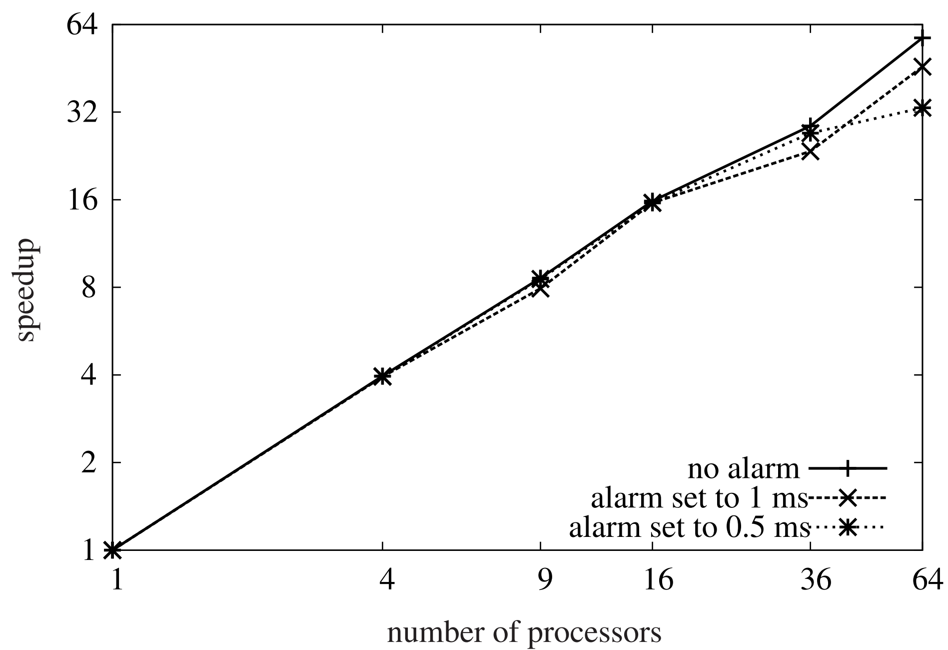


Figure 3.18: Speedup results for distributed Jacobi solver used for 2D temperature conduction simulation on 1000 × 1000 grid. Speedup curves are provided for the scenario without integrated framework as well as for the scenarios where *alarm* is set to 0.5 and 1 millisecond intervals. The overhead of the framework itself is negligible and speedup results are kept similar to the scenario without integrated framework.

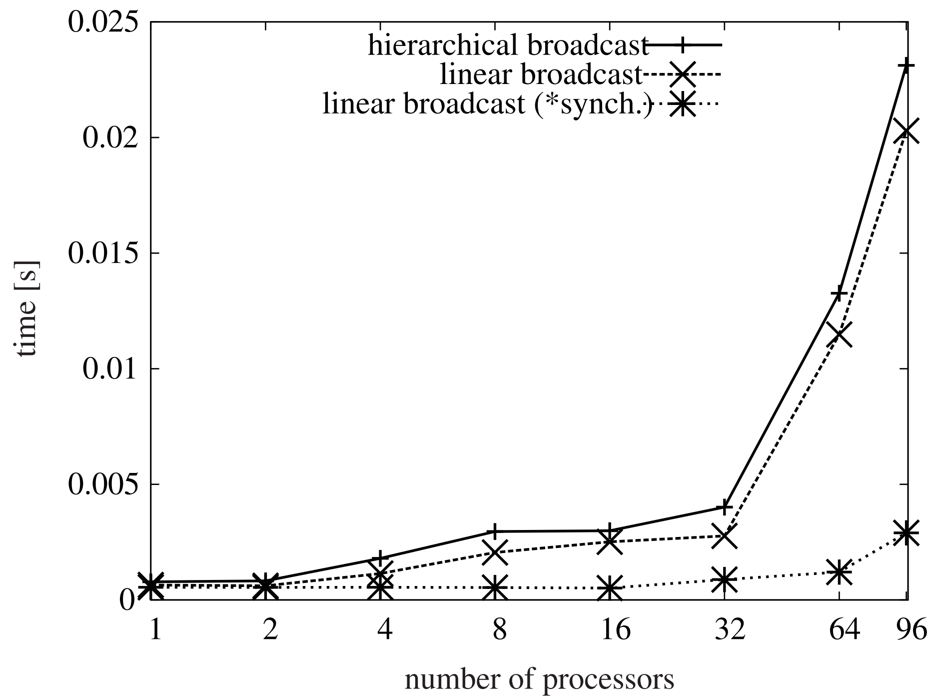


Figure 3.19: Measurements for linear vs. hierarchical broadcast algorithm on Inficluster at LRR-TUM (Opteron nodes, four AMD Opteron 850 processors 2.4 GHz each; equipped with two Gigabit Ethernet ports and with a MT23108 InfiniBand Host Channel Adapter card from Mellanox). Along the x-asis, the number of processors are presented; along the y-axis – the time needed to broadcast an update. It can be observed, due to delayed responses of some of the nodes, for the messages sent to more than 32 processes that there is more time needed for a message to arrive. However, if the running processes are enforced to synchronise at the beginning of the program execution, there is no significant delay noticed for 64 and 96 processors any more.

data transfer are very important, thus, they are tackled as well. Finally, hierarchical simulation approaches are compared in terms of their applicability in conjunction with the framework. These approaches are neither a part nor a responsibility of the framework, however, are supported and advised to be utilised whenever possible.

To integrate the platform into any application scenario, a few modifications of the code have to be made by the user. Since these modifications are only minor, all of them are listed:

- All the loop indexes which will be affected by the interrupt handler, i. e. interfaces to the framework, have to be declared global and both atomicity of their updates and prevention of the compiler optimisations which would lead to incorrect value references insured.
- The integrity of each user-defined 'atomic' sequence of instructions in the simulation code has to be ensured.
- All the variables related to the potential locking mechanisms, such as mutexes, semaphoras, etc. have to be declared global and "unlocked" within the signal handler. A user himself has to know and decide about his own variables.
- Critical sections have to be replaced by scoped locking mechanisms and corresponding variables should be declared global. Another option would be to block the desired signal in this part of the code, however, for guarantees for correct program execution, a user would have to look out of the OpenMP standard, thus, also out of the scope of what framework guarantees.
- A user has to be aware of his operating system and compiler related issues and choose an appropriate option supported by the framework.
- The calls to our send and receive functions have to be included in the appropriate places in the programs. However, a user himself has to instruct the interpretation of the data (e. g. in the receive buffers).
- Finally, he has to enable the regular checks for updates by including appropriate functions which will examine and change the default signal (interrupt) action and specify the time interval in which these checks should be made. These functions and their behaviour are system and compiler dependent. Most of the differences are documented and discussed within this chapter.
- Fast transfer of the data is ensured by selection routines which we provide. A user should be able to select part of the domain he wants to preview.

A user should also keep in mind that the prevention of compiler optimisations is done only to ensure the correct program execution in generic case. Some of

the optimisations can easily be enabled again by a user if he finds guarantees for the correct execution elsewhere. The variables which are manipulated within the signal handler are integers, often loop variables, and typically not numerous. If those variables appear elsewhere in the code, and a user would like them to remain local, they can be renamed in the segment of the code which is of interest (thus, declared global only there). The intention is to provide a generic, easy to integrate concept, thus, at several points a good trade-off has to be found. The concept is probably not the best suited for every application, however, is well-suited for many. It provides easily accessible, minimal invasive interface for a user whose field of expertise, or focus, may be different from real-time interactive computing.

Chapter 4

Application

4.1 Interactive Computing in Engineering Applications

Chapter 1 briefly described a wide range of the application fields using different concepts of, and tools for, interactive computational steering. It also explored the way experts in those particular fields, or more precisely in those particular scenarios, can profit from the adopted tools. The implemented, rather generic, framework described thus far is intended to support multiple applications. It must now be validated and evaluated through existing practical examples – namely, various engineering application codes. In this chapter, a few scenarios are presented in which the implemented framework itself has been integrated and tested.

Various conditions have significantly helped determine challenges, potential limitations, and areas of further investigation. These include a high diversity in terms of engineering fields, programming languages, code design and structure, user interaction interfaces; the overall system being distributed or local; the computation running in parallel, on multiple processing cores/nodes, or sequentially; etc. These conditions also support the exploration of possible additional requirements, which might be implemented in the future to provide additional features and extensions. A brief conclusion at this point is that the pool of different classes of applications is enormous, and each class, or subclass, has many rather specific characteristics – all of which would have to be supported by the framework. In order to achieve this, the user is still expected to provide some necessary information related to his specific case. The framework itself, as clarified in Chapter 3, cannot predict, for example, the names and types of variables to be steered, nor how to match the received simulation result from the message buffer to the visualisation data.

Before all the application scenarios are described in more detail, a brief classification of test cases in terms of diversity is provided, clarifying the challenges and the differences in integrating the framework. Namely, 2D heat conduction simulation, *AGENT*, and the *Bone* test cases have the user component and the

simulation component implemented as separate MPI processes, communicating the data over a communication network; all the *SCIRun* components typically run on a local machine.

Concerning the simulation itself, 2D heat conduction and the *AGENT* sequential computations discussed in this chapter use central FD and MOC approximation methods, respectively. However, FD approximation for a heat conduction simulation is also seen in the distributed parallel test case in Chapter 3, where the resulting system of equations is solved using the Jacobi solver. The applications tested within the *SCIRun* environment run multithreaded (particularly their concurrent parts, such as solvers of the systems resulting from a corresponding h-FEM approximation). The concurrent parts of the FCM-based *Bone* simulation component within the interactive environment run multithreaded as well (e. g., assembly function for the stiffness matrices). Moreover, the processes running for different polynomial degrees, as described later in more detail, are implemented as separate MPI slave processes, communicating the data to and from the user via one master node – a component of the framework itself. A distributed parallel stand-alone version of the solver has been developed with the intention of speeding up the computation and integrating it into the steering environment once the optimal parallelisation strategy is found.

Those applications also follow different design patterns. For instance, the *execution* design pattern in *SCIRun* is MVC, while the structural pattern is the module pattern. The *concurrency* pattern in *Bone* and *SCIRun* computations is *lock*, but on the level of the whole application in *SCIRun*, for example, it is *Scheduler*.

Programming languages used are: *C++* in 2D heat conduction and *SCIRun*, native *C* in *Bone* (*C++* only for the visualisation component), and *Fortran* in *AGENT*. The simulation component of all the application scenarios run on Linux systems, and in the case of the *Bone* environment, the visualisation component runs on Windows OS. For *simulations* running on Windows OS, only small tests have been done so far, yielding promising results in terms of successfully invoking a timer in small cyclic intervals to manipulate simulation specific variables, as described in Chapter 3. However, this scenario requires further investigation in more complex, "real-life" simulation test cases.

The different levels of challenges in integrating the framework from the user's point of view are discussed for each scenario at the end of every subsection of this chapter. The methods and results presented in this chapter were developed by the present author, and can be found in [102, 103, 108, 107, 109, 110, 104, 100].

4.2 2D Heat Conduction Simulation

4.2.1 Problem Description

To evaluate the concepts proposed in Chapter 3, as the first scenario, the framework was integrated into is a simple 2D heat conduction application. The first

version of the application was implemented for lectures[1] at the Chair for Computation in Engineering (TUM) in 1998 and has little scientific impact; however, it has offered a sound basis for the first integration of the framework into an existing interactive environment. Namely, it aims at exploring heat conduction in a domain, where different states and starting conditions tend toward stable equilibrium. In light of the alternating heat sources and their distribution in the finite domain, the application intends to estimate those equilibrium states for each setup.

4.2.2 Simulation

The simulation back-end is a C++ implementation that simulates heat conduction in 2D (described by Laplace heat equation) in a given region. After discretising the polygonal domain using a Finite Difference scheme for updating the values, one comes up with the well-known five-point stencil (Eq. 2.4). The system of linear equations is then solved using the Gauss-Seidel iterative method (Eq. 2.10, where all the elements of vector f are set to zero, according to the requirement of achieving the stable equilibrium state). Faster solvers, such as Conjugate Gradient, are not used since this application scenario serves only to test the framework, and it is more convenient to preserve the simplicity of the solver algorithm.

4.2.3 Interactive Visualisation

The graphical user interface for this application is implemented using the *wxWidgets* library [18]. OpenGL is used to visualise the gridlines of the discrete 2D domain with the assigned temperature (whose values appear along the vertical axes, pointed upward, as the "height" of the point). The range of temperatures (low to high) are mapped to the range of colours in the visual representation (green to red, respectively). This information is redundant, however, due to the already obvious "height" of each point.

4.2.4 Communication Pattern

The simulation and visualisation are implemented as separate MPI processes, communicating the update data (i. e., the position of the heat sources, boundaries, maximal number of iterations). The update events are caught by the user front-end process and stored in the event queue. Each item of the queue is an object with attributes representing a new x, y coordinate and the *type* (char) of action made by a user – representing, for example, mouse click, mouse drag, mouse release and similar events. Entries from the queue are sent to the simulation process and then removed from the queue. Once calculated, the result is sent back to the user. A simple communication scheme illustrating this is shown in Fig. 4.1. This exact setup was implemented in this work for testing purposes.

[1] the course Advanced Computational Methods 2 of the Master Program Computational Mechanics in 1998 [70]

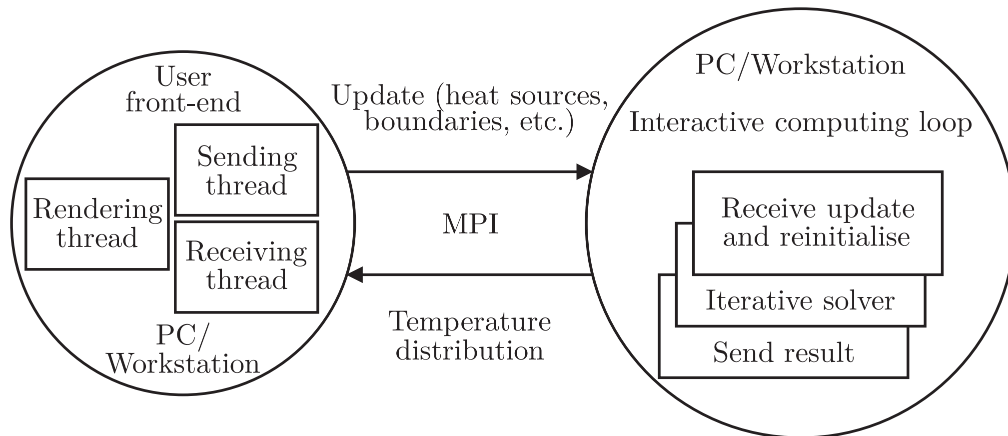


Figure 4.1: Heat conduction – 2D heat conduction simulation is running in the back-end. The solution is approximated using FD, and sent – as soon as it satisfies accuracy requirements imposed on the Gauss-Seidel iterative solver – to the user interface/visualisation front-end.

4.2.5 Initial Settings

The very first settings include grid generation parameters (i. e., the resolution in both spatial dimensions) as well as the solver parameters (i. e., error tolerance and maximal number of iterations for the simulation). The user specifies these parameters through a graphical user interface (Fig. 4.2). After providing these parameters, the user can define the boundary points of the domain, and the points (i. e., heat sources) with certain fixed values of the temperature set (Fig. 4.2).

4.2.6 User Interaction

A user can interact with the simulation during its execution time by adding, deleting or moving heat sources and boundary points, changing the maximal possible number of iterations or error tolerance.

4.2.7 The Framework in Action

Every time a user changed something, an event was added to the event (task) queue and a real-time signal was raised on the user front-end indicating the change. The signal handler sent the data about the collected events to the simulation through a message passing interface. (A similar mechanism could have been also implemented using two separate threads for user interaction and communicating the data.) The simulation, on the other hand, became aware of the message through the SIGALARM invocation. In the Gauss-Seidel solver algorithm, the loop index variables in both spatial directions were manipulated, restarting the computation. The beginning of the new computation required reinitialisation steps (e. g., initial guess for the solution was reset to a zero vector). If either the maximal number of iterations or the error tolerance was changed, these values

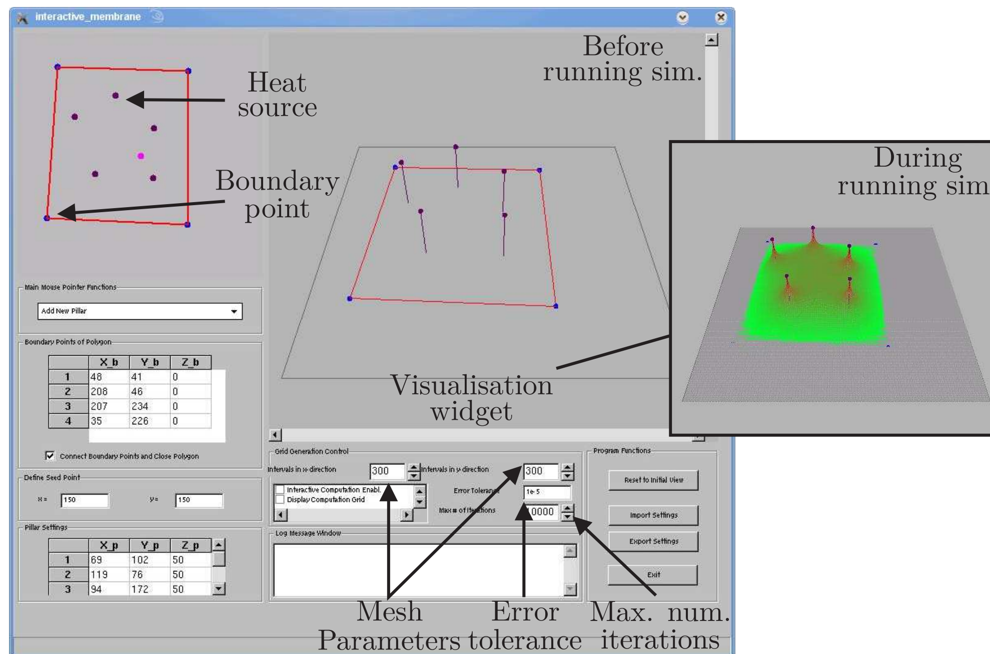


Figure 4.2: Initial settings: solver and mesh parameters, positions of boundary points and heat sources. Once a simulation is run, the overview of the temperature distribution is visualised after each iteration [70].

were reset for the new computation. The same occurred for the new positions of heat sources and boundary points. A reader may refer again at this point to Fig. 3.11 and the simplified pattern including MPI communication and probing routines.

As illustrated in Fig. 4.3, a calculation on a 300×300 grid immediately raised doubts as to the feasibility of instantly estimating the equilibrium state for points of the domain far away from heat sources. This was due to the short intervals between two restarts in the case of intensive user interaction. Here a hierarchical approach becomes necessary to provide instant feedback, which still satisfies the user despite its lower accuracy.

4.2.8 Hierarchical Approach

The hierarchical approach switched back and forth among several different grids, depending on the frequency of the user interaction. The principle was as follows: Initially, the desired grid was used for the computation. In the case of a user update, the simulation process recognised it, and as soon as it restarted the computation with the updated settings, it switched to coarser grids – with the level of coarseness dependent on the frequency of the user's activity.

In this particular test case, three different grids were used: the initial 300×300 grid; the intermediate one of 150×150 for a low pace of user interactions (e. g.,

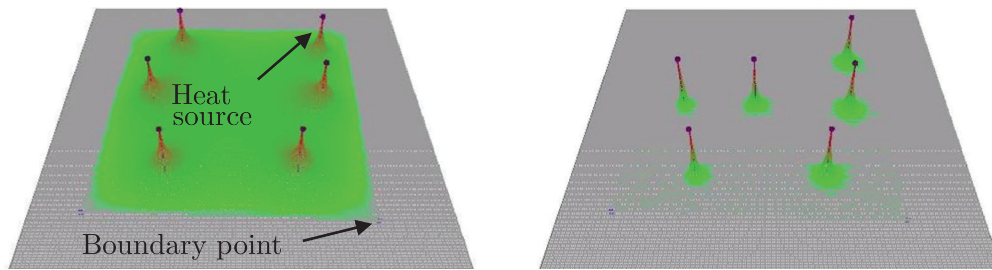


Figure 4.3: Left: the simulation running; right: the equilibrium state cannot be estimated due to frequent user interaction, thus, frequent restarting due to the interrupts.

adding or deleting heat sources or boundary points); and the coarsest of 75×75 (Fig. 4.4), for a very high frequency of moving boundary points or heat sources.

In general, at the initial point of interaction, one does not gain results as accurate as desired. However, the tendency of the running simulation in the overall domain can be easily and instantly observed, independently from the number and the rate of the applied changes. At some point the user may reduce his pace of interaction, trying to explore a "detailed", more accurate result. At last, when the current settings seem to satisfy the user's requirements, no more interaction takes place, and the stage of calculating even more accurate results is reached for very detailed analysis.

More specifically, as soon as the simulation at the back-end realised that there has been a front-end intervention in a certain time interval, it forced the actual computation to finish early and switched back to one of the coarser grids, depending on the interval (if less than 10 milliseconds, to the coarsest grid, otherwise to the intermediate one). The computation started then with the updated settings. When there was no user interaction, the interval before switching back from the coarsest to the intermediate grid was 10 milliseconds, and from the intermediate to the finest grid it was 100 milliseconds. These values were estimated by several experiments.

In this case, unfortunately, the results of the previous computations on the coarser grid were discarded. A multi-level method was used to speed up reaching the heat equilibrium on finer grids, where the previously calculated results were reused. This is discussed in more detail further on.

An upsampling algorithm was employed to not waste the results of computations gained during the highly interactive mode. The results on the coarsest grid were taken into account when switching to the finer one. The hierarchy of the grids offers benefits analogous to a multigrid algorithm, albeit with a different approach. The multigrid algorithm accelerates the convergence of a basic iterative method by global correction from time to time through solving a coarser problem (i.e., descending to the coarser grids and calculating an error). In contrast, our scheme (when the user interacts) started with the solution on the coarsest grid

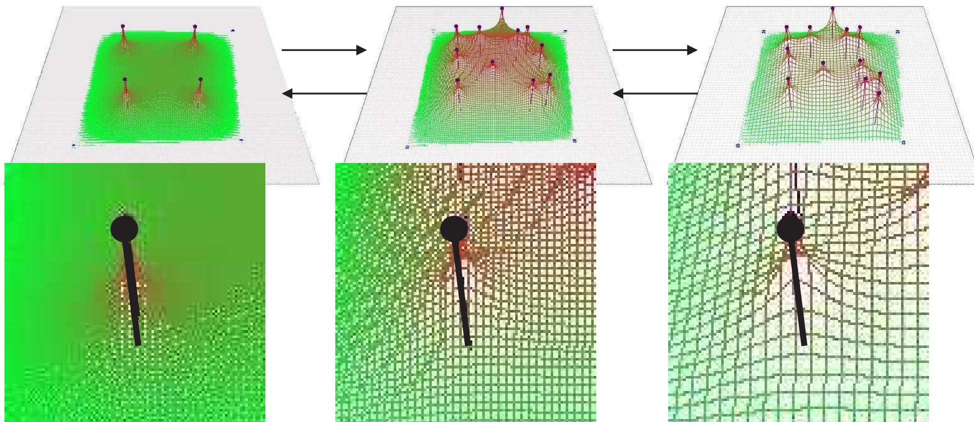


Figure 4.4: Switching between different mesh parameters. In the case of a user starting moving boundaries, or heat sources, it was instantly switched to the coarsest grid; in the case of adding/deleting heat sources or boundary points, it was immediately switched to the intermediate grid. When a user stops interacting, it was gradually switched from one grid to the next finer one, until the finest one is reached.

and only used the result we gained as an initial guess of a result on a finer one [102, 103] (Fig. 4.5). This approach provides a sensible preview to the user due to the smoothness of the solution in most of the domain. Once the desired setup is found, the simulation can be executed for it, starting from scratch with a zero vector as an initial guess.

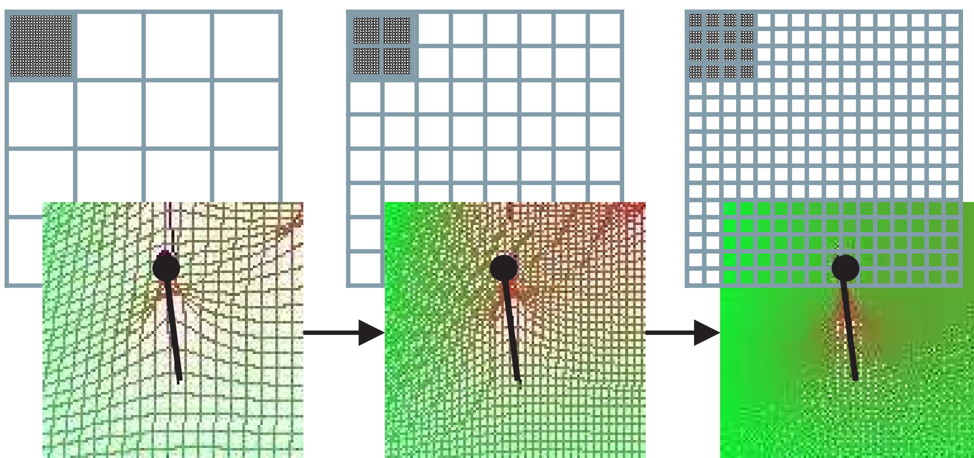


Figure 4.5: Multi-level approach.

4.2.9 Results

Due to the analogy of the Gauss-Seidel to the Jacobi scheme, the overhead graphs for the framework in terms of its execution time are comparable to those for the simulation with the sequential Jacobi solver algorithm presented in Fig. 3.17.

The results, presented also in [102, 103], show clearly that the convergence rate is higher with the multi-level approach. For the examples of the settings tested, the number of iterations needed for convergence both on the intermediate and the initial grid can be improved by a factor of 2. This held for the initial 300×300 grid and several different heat sources $P_i(x_i, y_i)$ and boundary $B_j(x_j, y_j)$ points with corresponding ordered pairs (x_k, y_k) of x- and y-axis indices respectively and the error tolerance set to $e - 05$. Nevertheless, measurements also showed that the variation of the solution on the finest grid is around 4.5% compared to the intermediate grid, and around 14.6% compared to the coarsest one.

4.2.10 Effort to Integrate the Framework

As a conclusion to this application scenario, a brief summary is presented below of the steps in integrating the framework:

- The framework function for overriding the default SIGALRM signal handler was inserted in the simulation code (see MOD. 3 in Appendix).
- The loop variables manipulated within this function to skip the redundant computation were declared global, *sig_atomic_t* and *volatile* (see MOD. 1 in Appendix).
- The signal handler function was edited according to the needs of a specific code and compiled together with the rest of the code.
- The implementation of the signal handler function in this case consisted of probing a message from a user, manipulating the loop indexes, if needed, and specifying the hierarchy depending on when the last update occurred (see MOD. 4 'for hierarchical approach' in Appendix).
- Depending on the hierarchy, the program had to switch to a pointer to the corresponding grid (see Gauss-Seidel solver function in Appendix). (The hierarchical approaches in general are not part of the framework itself, but rather a concept well supported by the framework, and thus these should not be considered as additional effort in integrating the framework.)
- Reinitialisation had to be done at the beginning of the interactive computing loop. When switching from a coarser to a finer grid, this was done by interpolating values from the coarser one as an initial guess on the finer one. When switching to the coarsest grid, the initial guess was a zero vector.
- The appropriate communication function calls were employed (see MOD. 2, MOD. 4, MOD. 5 and MOD. 6 in Appendix).

From the user's point of view, to enable the framework functionality for interrupting the simulation was very straightforward, and took a couple of hours at most. Implementing the hierarchical approach (which is technically not a part of the framework, but is worth mentioning) was more time consuming (in terms of a couple of working days). Here, an optimal automatic detection when to switch from one hierarchy to another (based on time intervals in-between two user interactions) had to be enabled, which required numerous experiments.

4.3 AGENT Software

4.3.1 Problem Description

While tremendous advances in computer hardware and computational methods enable a variety of complex physics problems to be analysed and simulated with increased accuracy, these experiments still require a significant amount of time. This refers to the numerical modelling of the particle transport methods as well, which has even been abandoned in the past due to computer power, speed and memory limitations [90]. In nuclear engineering in particular, both high-accuracy and real-time simulation of big reactor cores – diverse in material and other properties – is still beyond reach. Researchers in this field, however, aim to have more impact on this time-consuming process, especially in cases when certain acceleration methods allow for faster convergence towards a solution with little to no loss of accuracy.

4.3.2 Simulation

AGENT (Arbitrary GEometry Neutron Transport) solves the neutron Boltzmann transport equation, both in 2D and 3D, using the Method of Characteristics (MOC) [91]. To approximate the problem, the continuous domain has to be discretised. First, a number of parallel rays (i. e. characteristic lines, see Subsection 2.2.1: *Method of Characteristics*) are generated in a discrete number of directions to represent neutron trajectories. Along those, as shown in Subsection 2.2.1, the differential operator in transport equation reduces to the total derivative. Second, the regular geometry mesh of the reactor domain is generated and the intersections with the parallel rays is found. Discrete scalar flux is then iteratively calculated within the sequence of multiple nested loops.

Modelling the exact geometry of a reactor system requires a fine spatial discretisation to keep material properties on each discretisation unit constant, and thus to provide an accurate solution to the problem. On the other hand, a good initial guess for the scalar flux always tremendously speeds up the convergence, and this property is used to considerably profit from the implemented interactive steering concept [110].

4.3.3 Interactive Visualisation

To examine the suitability of the chosen parameter set during the run time, a user must be able to understand the properties of the current solution. Therefore, the simulation periodically outputs the current result for visualisation. The visualisation is performed on a local machine, using the *ImageVis3D* volume rendering tool [74].

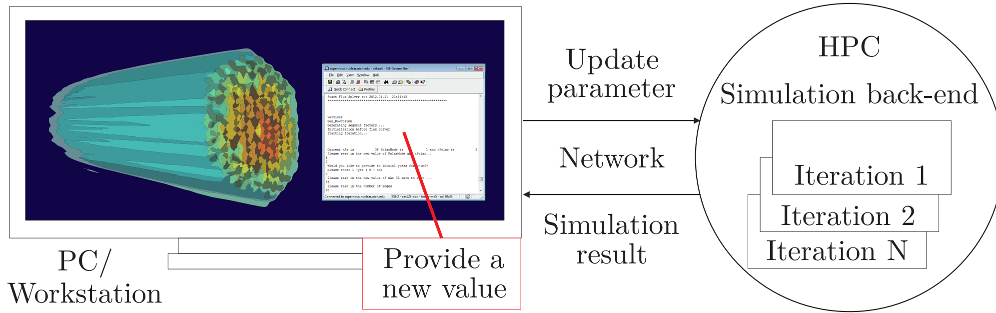


Figure 4.6: *AGENT* simulation back-end runs on the remote cluster; visualisation front-end consists of a simple console interface where a user can interrupt the simulation and modify the desired parameters.

4.3.4 Communication Pattern

The steering environment consists of the simulation, which runs on a remote server, and the visualisation component, which runs locally. The visualisation system queries the server data. The server maintains a list of available simulation results, and the client connects to the server, queries the data and downloads it to the local machine where it is visualised. This approach can be improved for larger dataset, however the current solution was sufficient for the data sizes presently used for running the *AGENT* software [110].

4.3.5 Initial Settings

A range of parameters has to be specified as the initial settings, such as accuracy, number of iterations, geometry grid resolution, number of azimuthal angles, ray separation and initial guess for scalar fluxes. The corresponding set of variables in the program defines the interface to the framework.

4.3.6 User Interaction

The user can intervene at any point with the running simulation via a simple console interface. Typically, after previewing an intermediate result achieved by changing a simulation parameter, he might want to further optimise the experiment. When a keyboard event signals the simulation to stop, the new values of the desired parameters must be provided via the same console interface. The

updates are automatically applied within the restarted computation, or, more precisely, at the beginning of the interactive computing loop.

4.3.7 The Framework in Action

The framework aimed to provide an instant response to the change of parameters made by a user in the *AGENT* simulation, ensuring that the regular course of the computation could be interrupted at any point. This was accomplished in this case via user-generated keyboard events. The default behaviour of the SIGINT signal was overridden so that certain simulation variables were automatically assigned a value outside of their domain. These variables also determined the range of a few loop indexes in the code, such as the number of planes and the discrete number of parallel discretisation rays – so-called azimuthal angles. This manipulation of the specific variables was done, as always, within the signal handler. Afterward, control was given back to the simulation, which then continued from the state saved at the previous interrupt-point, skipping most of the then outdated computations.

The framework directly supports the most straightforward procedure of steering the variables which do not affect the dimensions of the rest of the data – such as the maximal iteration number or the convergence criteria. In principle, the concept of the framework is flexible: a user can actually choose whether to proceed with the running iteration (without manipulating loop variables), and only then to start anew with the updates, or to skip the rest of the iteration. On the other hand, the support for steering of the discretisation parameters is categorised as a hierarchical approach.

4.3.8 Hierarchical Approach

This category typically refers to steering the discretisation parameters used in numerical modelling and is thus considerably more challenging. For the reasons mentioned in the previous subsection, the hierarchical approach assumes that the user still wants to preserve and reuse some of the values from the previous calculation as an initial guess for the solution in order to accelerate the convergence. It is distinguished between the two subcategories.

The first subcategory refers to the discretisation parameters: the numbers of azimuthal angles, polar angles and boundary edges or ray separation. In the *AGENT* numerical model, for instance, the number of azimuthal angles represents the discrete number of directions in which neutrons move; thus, an increase in this value increases accuracy, but prolongs the computation. One actually accelerates the convergence process with no loss of accuracy by running the simulation first for smaller numbers of azimuthal angles and then using the calculated scalar fluxes after a certain number of iterations as an initial guess for the solution on a higher resolution grid (Fig. 4.7).

The implementation of the steering pattern in *AGENT* requires more effort if the resolution of the geometry mesh is manipulated, and thus the values of the

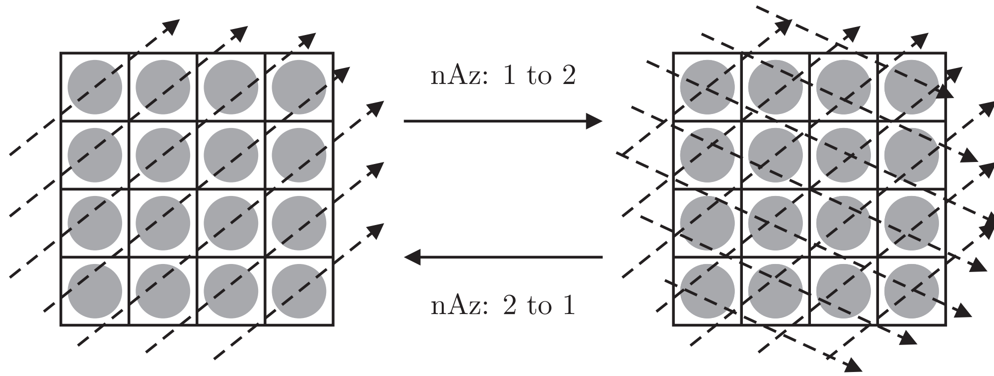


Figure 4.7: Switching between different numbers of azimuthal angles. The small numbers (1 and 2) are shown only in order to simplify the picture.

coarser mesh must be interpolated to the finer mesh as an initial guess. This is because all variables and fields must be properly reinitialised. However, it is estimated to be very beneficial and is therefore planned to be carried out at the University of Utah's Nuclear Engineering Program (UNEP) in the near future.

Future research in this direction shows promising potential [110] based on the results already obtained by experimenting with the number of azimuthal angles and on the fact that the framework has already yielded excellent results supporting conceptually similar algorithms in other scenarios, such as the multilevel approach in the 2D heat conduction simulation in Section 4.2 or other kinds of hierarchical approaches described later in Section 4.5.

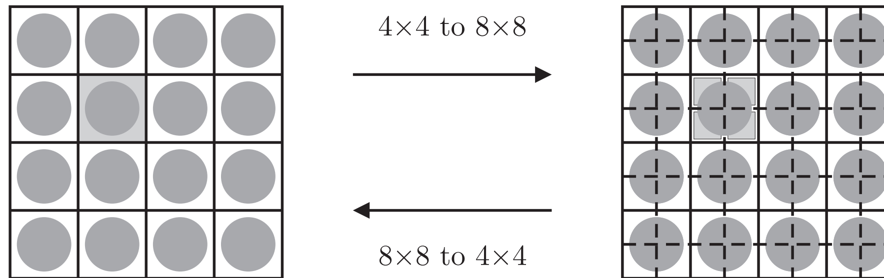


Figure 4.8: Switching between different geometry mesh parameters. In order to re-use the previous calculation as an initial guess, the values would have to be inter- (when switching from finer to coarser mesh) or extrapolated (when switching from coarser to finer mesh).

4.3.9 Results

For this particular scenario, one solver iteration (i. e. one update which can be visualised) may take longer than real-time depending on the predefined problem size and convergence criteria for the solver. Nevertheless, it is most important to be able to immediately apply updates during the execution time and to use the

results from previous iterations to accelerate the convergence. To estimate the effectiveness of the framework, the measurements are made with regard to the simulation's responsiveness to user interaction – that is, the time needed to restart the simulation – as well as the CPU-time/solver convergence acceleration from changing the parameters during the simulation run time. The measurements in this subsection are published in detail in [110, 104]. Here, most of the published results and conclusions are given in brief.

Time to restart the simulation

To evaluate the effectiveness of the framework, as reported in [110], the tests were performed for a fine discretisation along the z-axis based on 20 horizontal planes, each of which is further discretised by a 300×300 geometry mesh and 36 azimuthal angles (parallel rays along which the neutron transport is simulated). The loop variable representing the number of planes (*nPlanes*) was manipulated first in the signal handler, setting its value to -1; thus, the time needed to restart the iteration at any point during its execution was reduced from 500 seconds – the execution time of one, outermost iteration for this choice of simulation parameters – to 25 seconds. Manipulating further the number of azimuthal angles (*nAz*), this restart took only few seconds; whereas, by manipulating both *nPlanes* and *nAz* it took a second or less, no matter at which point a user performs a change.

As the measurements indicate, for the currently used data input, there is no need to manipulate more than the two aforementioned simulation variables to instantly start the computation anew. However, in the case of larger problem sizes, it might be desirable to manipulate more loop-range parameters, such as the number of energy groups to prevent any delay in the restart [103]. The framework allows for additional variables to be added by a user for steering.

Experiments

Since the *AGENT* code solves the neutron transport equation using an iterative process, two convergence criteria are used. The first is the value of relative difference for the multiplication factor between iterations (*kdiff*), and the second is the maximum relative difference of zone flux for all zones, i. e. with the same material properties. It is now considered how steering different parameters can accelerate convergence towards the solution.

The experiments were done after the integration of the framework at the UNEP. In the experiment described in [104], a user starts the simulation with a non-optimal set of initial parameters; thus the convergence follows a slow trend. If the simulation is interrupted during a particular iteration, the next experiment could be started with the same set of parameters by conserving the same geometry mesh but refining the MOC resolution parameters. The resulting solution will improve toward the best estimate and also in a shorter computation time. Fig. 4.9 shows the values for the solution (i. e., $phi_{diff,max}$, scalar neutron flux). Namely, in contrast to the previous non-interrupted simulation where at the 600th iteration the convergence criteria is not satisfied yet, the interrupted scheme reaches the

convergence criteria around the 500th iteration [104], also achieving the more accurate value of *kdiff* [104].

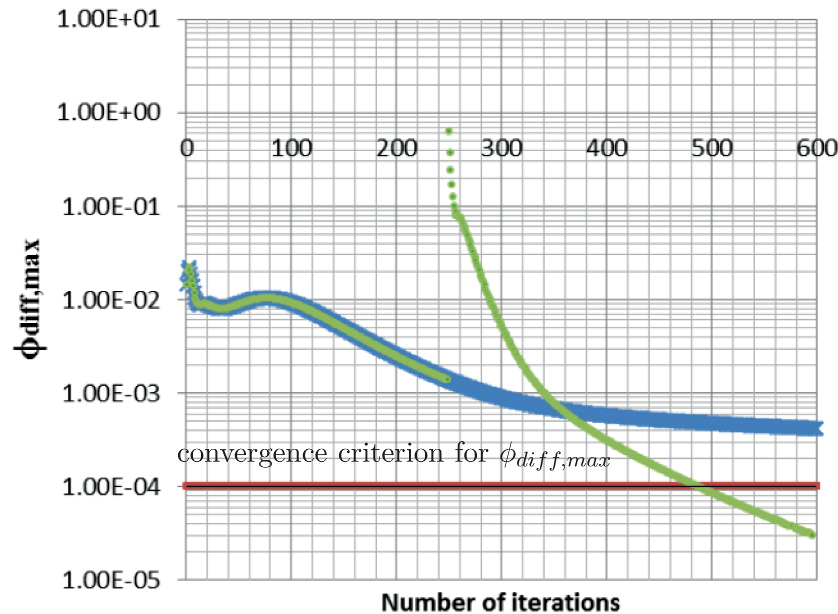


Figure 4.9: The results of the experiments done at the UNEP (Hermilo Hernández, Tatjana Jevremović). The outcome of this collaborative work is published in [104]. The picture itself shows that choosing an optimal set of parameters, the convergence of the solution can be significantly accelerated. Namely, in the 600th iteration the convergence curve does not meet the limit line yet. If, however, the simulation is run with the same set of parameters, then is interrupted in the 249th iteration, when a more optimal set of parameters is chosen, the solution converges already around 500th iteration.

AGENT simulation steering reducing total CPU time

Another experiment done at the UNEP, described in [104], showed the effect of the interruption early during the *AGENT* simulation of the *TRIGA* research reactor. The *AGENT* iterative process started with the low-resolution parameters, which in themselves – as proved by the previous experiment – do not meet the convergence criteria, but rather create an initial solution estimate. The interruption was introduced after a certain number of iterations. Two independent cases based on MOC resolution were analysed after the interruption: medium resolution and higher resolution. As reported in [104], the low-to-medium interruption procedure in this experiment provides 15.6 % of the CPU-time saving. This gain in time increases with the low-to-high interruption procedure which consumes around 22 % of the CPU time with respect to a high-resolution level computation.

4.3.10 Effort to Integrate the Framework

From the user's point of view, the integration of the framework was straightforward and also not time consuming. After examining the initial AGENT code, deciding which variables to register within the framework, and writing reinitialisation routines, it took several hours to couple everything together and enable visualisation after each iteration. The greatest effort was required for the proper reinitialisation of variables before the new computation starts. This was a kind of input which, like always, had to be provided by the developer of the simulation code and method. A detailed and more technical overview follows:

– The framework function for overriding the default SIGINT signal handler was inserted in the simulation code.

– The C wrappers for *signal* function, as well as for clearing the previous signal action, had to be implemented, compiled and linked with the rest of the code.

– Those functions had to be called from the main program.

– A signal-handler (Fortran) subroutine had to be edited according to the needs of a specific code, then compiled together with the rest of the code.

– The loop variables manipulated within this Fortran subroutine had to be visible in all the other subroutines using it in order to skip the redundant computation.

– The implementation of the signal handler function in this particular case consisted of manipulating the loop indexes and providing new values for solver and discretisation parameters.

– Depending on the parameter changed, the appropriate reinitialisation routines had to be called before the new computation started.

– The reinitialisation of some light parameters, such as the maximal number of iterations or convergence criteria was straightforward and effortless.

– Steering discretisation parameters required more effort, depending on whether the parameter affects the solution size. For the number of azimuthal angles, a few reintialisation steps had to be taken, which were similar to actual initialisation subroutines and thus did not require major effort.

– The geometry mesh parameters require interpolation of the values from one mesh to another and thus more user effort. However, this has yet to be tested.

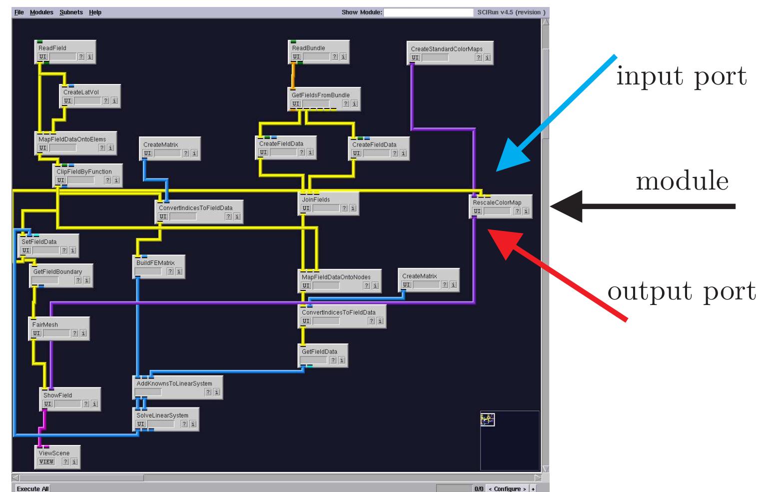


Figure 4.10: SCIRun network of modules, connected via input/output ports. For each of the modules, a small widget with an interface to different module-related parameters can be open.

4.4 The SCIRun Software

4.4.1 Problem Description

SCIRun [93, 124, 134, 125] is a Problem Solving Environment intended for interactive construction, debugging and steering of large-scale, typically parallel, scientific computations [150]. It is a modular, easily extendable software package based on dataflow programming, and it provides an efficient and comfortable environment for exploring different computational models.

4.4.2 Simulation

A *SCIRun* simulation is designed as a network of modules connected via input/output ports (Fig. 4.10). It allows for adding new modules and easily modifying each of them without affecting others. Object-oriented *SCIRun* code uses the Model-View-Controller design paradigm (Fig. 4.11), where the Controller entity – so-called *Scheduler* – is in charge of all other modules and their priority of execution. Storing one module in the execution queue causes all modules with input ports attached to that module to be also scheduled for execution.

The simulation test cases for the integration of the framework are: (1) a simulation that facilitates early detection of acute heart ischemia and (2) two defibrillation-like simulations on two different domains – the first being a simpler, homogeneous cube domain, and the second an inhomogeneous human torso.

Heart ischemia model

This application aims to generate a quasi-static volume conductor model of an ischemic heart based on data from actual experiments [155] to help early detection of heart ischemia. The modelling pipeline requires the generation of experiment-specific models of the myocardium, based on the magnetic resonance (MR) scans of a dog heart. The known values are extracellular cardiac potentials as measured by electrodes on an isolated heart or with needles inserted. The potential difference between the intracellular and extracellular space is not the same for ischemic and healthy cells, which causes so-called injury currents to flow within the two spaces. These potential differences translate to so-called ST shifts in the electrocardiogram (ECG) [155]. Using data from experiments, a network of modules can be constructed within SCIRun to simulate and then render a model of the transmembrane potential of a dog's myocardium.

Simplified defibrillation simulations

A simple defibrillation-like example is first examined: a simulation of the electrical conduction on a homogeneous cube domain with two interactively placed electrodes. Each of the electrodes is assigned a conductivity value, which can then be changed within the interactive computing loop.

For a given pattern of source activation, one of the bioelectric field problems cardiologists examine is the spreading of the electric activity through the rest of the domain. Such studies are used when investigating internal implantable defibrillator designs, as discussed in Chapter 1. Constructing the computational model of a patient's body and mapping appropriate conductivity values over the domain, doctors can accurately estimate how activity generated in one region would be remotely measured in another region [173].

4.4.3 Interactive Visualisation

There are several visualisation modules incorporated in *SCIRun* that can be used to preview the result after the defined number of iterations. This parameter may be specified in the *SolveLinearSystem* module. For instance, one can extract isosurface from the field data and then map a colour from one of the colour maps to an isovalue. In the case of an interrupt in any of the modules (which are typically higher in the execution hierarchy), the visualisation modules are also scheduled for execution. From the framework point of view, it is important that the execution of the visualisation modules in the presented simulation scenarios last several seconds. Thus, it is necessary to remove them from the schedule once a user interaction takes place.

4.4.4 Communication Pattern

Communication within simulation and visualisation components is implicitly established via shared memory for all the test cases described above. *SCIRun* also supports remote execution of a (multithreaded) simulation via TCP/IP socket

communication. To profit from the framework in this case, this communication pattern would have to be replaced with the framework non-blocking routines.

4.4.5 Initial Settings

The initial phase reads in the data and sets all necessary parameters related to different modules – ranging from values located in various simulation-specific fields, to the colour panel settings, etc. The user interface to these values is provided separately for each corresponding module.

4.4.6 User Interaction

Concerning the solver parameters, different solver options may be used: Conjugate Gradient (CG), Biconjugate Gradient (BCG), Jacobi, Minimal Residual (MINRES). One can also activate pre-conditioners, such as Jacobi, and influence error tolerance, the maximal number of iterations, the frequency of emitting the visualisation, etc.

In the case of defibrillation-like simulations, two electrodes with two input conductivity fields should be placed; thus, the corresponding module and its variables can be accessed for each electrode and modified interactively.

In addition to manipulating solver-related parameters – for the resulting system of the linear equations – along with conductivity-related parameters, it is possible to switch between different mesh resolutions in h-FEM approximation. This allows the user to preview the solution on a coarser grid and switch to the finer one (starting from scratch) once the user is satisfied with the current setting. It could be advantageous to run a separate process with higher resolution in the background, then to update the solution on the coarser grid as soon as the user stops interacting (or after a certain number of iterations).

4.4.7 The Framework in Action

SCIRun provides an optimal software environment for integrating the aforementioned interactive computing framework[2]. The dataflow model (Subsection 2.6.2) allows for triggering only the re-execution of the necessary modules due to user updates. In this an already mature and sophisticated environment for computational steering, as pointed out in [100], the goal was to have real-time feedback for even more time- and memory-consuming simulations. To re-execute a module during user interaction, it was essential that the redundant, often long, computations were skipped. Hence, any module previously scheduled for execution had to be canceled at this point. However, the execution of all modules following the modified one in the pipeline had to be re-triggered. These features were provided

[2]The work related to SCIRun PSE was made possible in part by software from the NIH/NIGMS Center for Integrative Biomedical Computing, 2P41 RR0112553-12. It was accomplished in winter 2011/12 during a three-month research visit of the author to the Scientific Computing and Imaging (SCI) Institute, University of Utah.

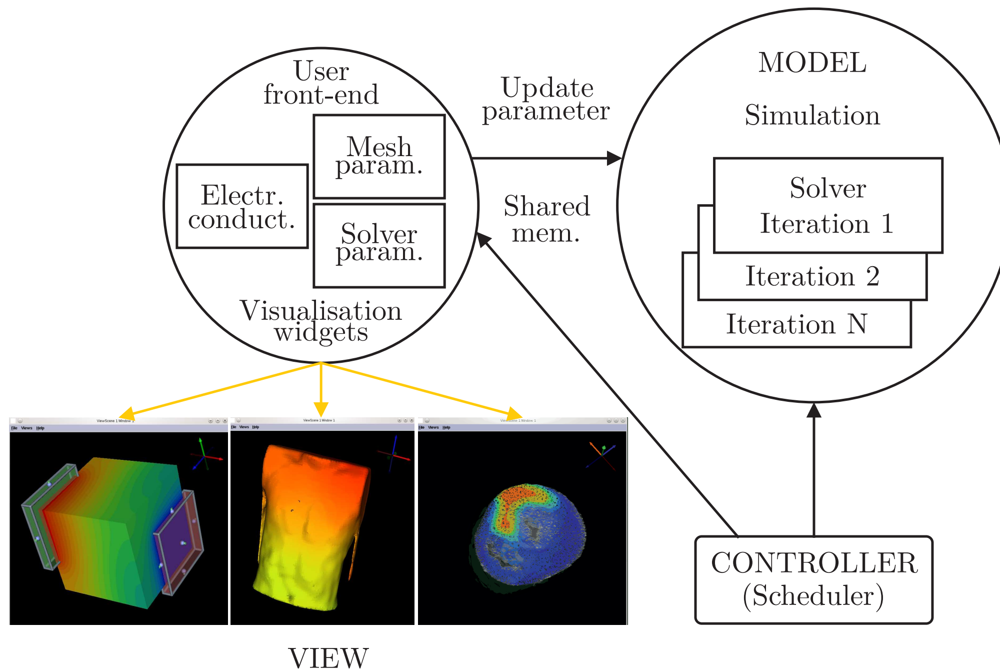


Figure 4.11: SCIRun GUI – a visualisation widget and user interfaces – one for each module on the front-end; one of the three simulations is running also locally – a tool for early detection of heart ischemia, or defibrillation-like simulations on homogeneous cube and a human torso.

via the integrated framework. The framework was tested on all the aforementioned simulations to evaluate user response for different overall execution times, orders of module execution, changes in chosen parameters, etc.

Interrupting the solver

In the network of modules created in this simulation, the most computationally expensive step was typically the execution of the *SolveLinearSystem* module. Thus, the first challenge for the framework was interrupting it as soon as a user made a change via the user interface. To achieve this in the iterative solver algorithm for the system of linear equations, the maximal number of iterations (normally a user interface variable) was replaced by a globally visible variable registered within the framework. This value was then manipulated in the signal handler, as described for all the other test cases as well. Afterward, the execution of this module had to be re-scheduled with the updated settings. However, after the integration of the framework, the previous interrupted execution of the same module had to be finished in a clean way.

To achieve this, all other system components had to be notified about the return from a module-specific *execute* function. First, the input and output ports – which had been opened by the previous *execution* call – had to be closed and re-opened, so that the *SolveLinearSystem* variables are properly reinitialised. Second, max-

imal number of iterations had to be reset along with some other user-interface variables which would have been automatically reinitialised without the interruption. One such variable, for instance, was determining whether the partial solutions should be emitted. Finally, the *Scheduler* had to be explicitly informed about this new execution and had to confirm that it had stored the identification of a task to be executed.

Interrupting the scheduler

To emit the partial solution after each iteration, as mentioned before, scheduling the executions of several visualisation modules takes place, prolonging the time to finish one iteration by several seconds. Thus, after an interruption of one solver iteration achieved by the framework, the preview of old results had to be canceled. In this way, the execution of all modules that would happen after the current module had to be aborted. This was achieved by enforcing an exception in one of the classes so that it was automatically caught where needed in all other modules.

4.4.8 Results

The overhead of the framework was tested for all of the three simulation scenarios described above (see also [100]). Different update-checking intervals were considered along with different problem sizes and solvers of linear systems of equations.

For the electrical conduction simulation on a human torso with inserted defibrillators – with the mesh resolution ($50 \times 50 \times 75$), in the case of the shortest *alarm* interval (i. e. 1 millisecond) – the overhead caused by the framework is around 15 % (Fig. 4.12). By prolonging the interval to 2 or 5 milliseconds, the overhead is reduced to around 5 and 2–3 %, respectively. With *this* increase of the interval, however, an end user does not intuitively notice the difference in terms of the simulation response.

In the case of electrical conduction simulations on the homogeneous cube domain (with two inserted electrodes), tests have been done for two problem sizes – mesh ($32 \times 32 \times 32$) and ($64 \times 32 \times 32$) – and again no more than 5 % and 10 % overhead is observed for these two calculations, respectively (Fig. 4.13).

For the simulation based on the heart ischemia model, even less overhead is apparent – namely, no more than 5 % in all cases. (One must also consider that for such small overall execution times, 1–2 % overhead can also be assigned to the cache coherence and measurement precision issues).

Therefore, in conclusion it is recommendable to experiment with different *alarm* intervals for a specific simulation, in particular if one observes that the execution time has been significantly extended.

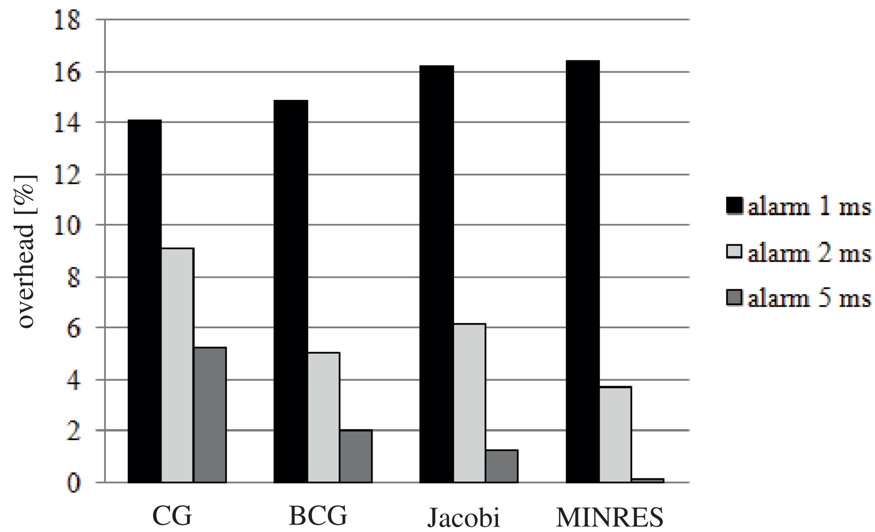


Figure 4.12: Overhead of the framework (vertical axis – in %) for the simplified defibrillation simulation on a human torso, shows that an optimal interval for setting the SIGALARM signal to occur is rather 2 or 5 milliseconds, not 1 millisecond, due to the higher overhead in the latter case than desired. Tests are done for different solvers represented on the horizontal axis.

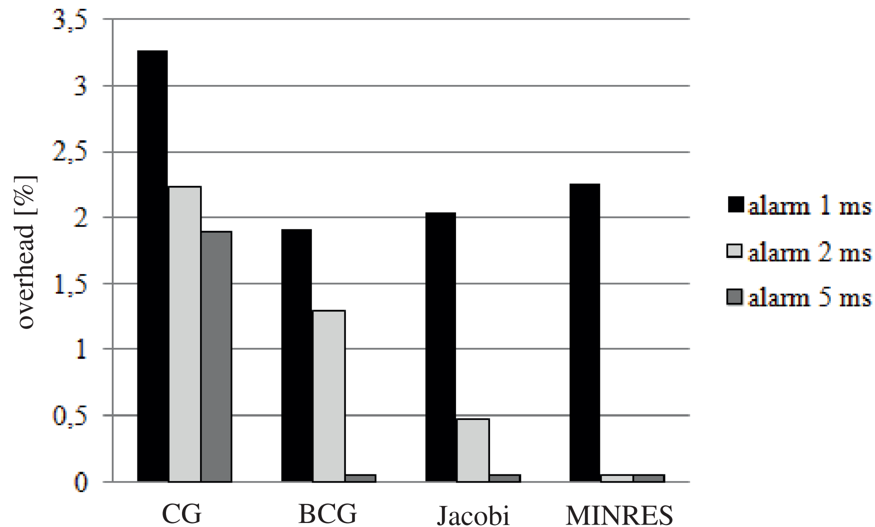


Figure 4.13: Small overhead (vertical axis – in %) for the simplified defibrillation on a homogeneous cube – mesh ($32 \times 32 \times 32$). Tests are done for different solvers represented on the horizontal axis.

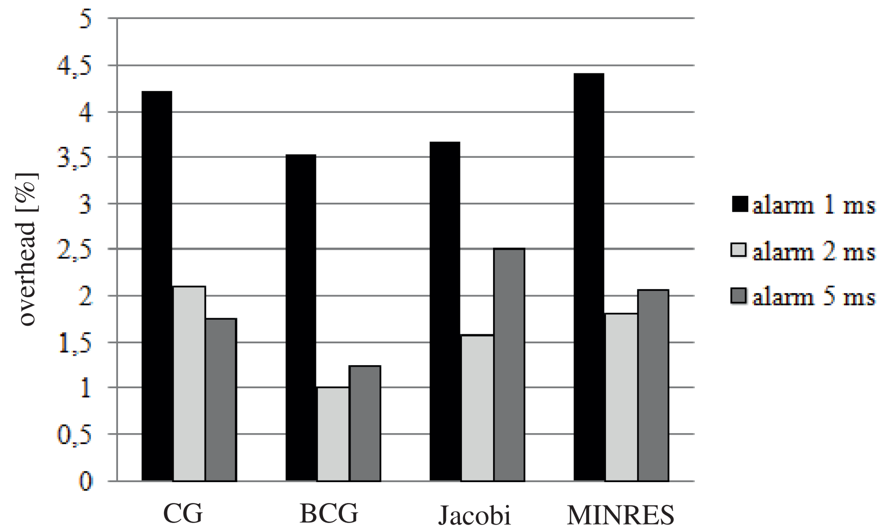


Figure 4.14: Small overhead (vertical axis – in %) for the tool for early detection of heart ischemia. Tests are done for different solvers represented on the horizontal axis.

4.4.9 Effort to Integrate the Framework

For a user to integrate the framework into the *SCIRun* code, the main challenge was related to finding a way to automatically re-trigger the execution of all the needed modules when the user makes a change of a parameter value. This required good understanding of a Model-View-Controller pattern within the code. It took almost no time to register both the variables whose update can be recognised and those which need to be manipulated within the framework to interrupt the execution of the modules of interest. More detailed description of the code modifications is provided below.

- The framework function for overriding the default SIGALRM signal handler was inserted in the simulation code, and the SIGALARM was set to occur in predefined intervals.

- The loop variables manipulated within this function in order to skip the redundant computation were declared global, *sig_atomic_t* and *volatile*.

- The signal-handler function had to be edited according to the needs of a specific code and compiled together with the rest of the code.

- The implementation of the signal-handler function in this particular case consisted of checking if an update had occurred and manipulating either the loop indexes or the variable used as an indicator for the Scheduler to cancel the outdated tasks.

- Variables in the loop of an iterative solver had to be replaced with ones registered within the framework which could be checked or manipulated, respectively, within the signal handler.
- Reinitialisation had to be done at the beginning of the new computation.
- Finally, the previously described steps had to be taken to cleanly re-execute the interrupted module and cancel the execution of Scheduler jobs
- In a distributed environment, the communication library calls would have to be replaced with the corresponding non-blocking framework function calls; however, this scenario has not yet been tested.

4.5 Virtual Planning of Hip-Joint Surgeries – "Bone"

4.5.1 Problem Statement

To determine the optimal implant design and position, an orthopaedic surgeon would like to experiment with a model of a bone based on computed tomography (CT) scans and estimate the response of the bone to different implant positions and designs and to moderate load application – for instance, walking up or down stairs. The tool was developed for this purpose within a collaborative project of the International Graduate School of Science and Engineering (IGSSE) – as previously mentioned in Sections 1.4.2 and 2.8. It consists of a graphical user interface and a visualisation widget for the bone stresses and strains on the front-end, and the simulation of these on the back-end [66, 67, 178, 179, 177, 65].

4.5.2 Simulation

The simulation kernel is based on the geometry models of the femur (i. e. thigh bone) constructed by CT/MRI-data. For the numerical approximation, the Finite Cell Method (FCM) – a variant of the high order p-FEM code with fictitious domain approach – was used, as proposed in [72]. This method supports complicated geometries or multiple material interfaces without an explicit 3D mesh generation. This is a very important property for interactive computing, since this typically time-consuming step would otherwise have to be done anew for each new configuration. The basic idea is an extension of the physical domain up to the boundary of an embedding domain, which can be meshed more easily.

Interactive computing loop

At the beginning, the femur voxel information is generated on the visualisation side. The model is based on the quantitative computed tomography (QCT) scans, which indicate bone strength via models like those presented in [181, 176]. To approximate the solution of the governing equations, the rectangular domain embedding the entire femur is generated. The domain is then divided into cells of the same size. The polynomial degree of the shape functions p and the number of voxels in each direction are read from the user input file. The computational

domain is kept fixed during the entire run time of the simulation as the time-consuming discretisation is done only at the beginning, making it efficient for interactive computing. The computational effort is, however, now shifted from the mesh generation to the calculation of stiffness matrices (i. e. integration) in order to capture well the domain boundary. To advantage, for each voxel the constant matrices needed for further efficient calculation can be *pre-integrated* analytically [179].

Driven by external forces *f*, a deformed solid is governed by the well-known equations from static elasticity theory. Applying FCM as an approximation method yields a linear equation system 2.5. Due to the often poor condition number of the system, standard iterative solvers fail to be efficiently deployed, thus, a special treatment allowing for both the design of sophisticated solvers as well as for advanced parallelisation strategies is applied. That is to say, a direct solver with hierarchical concepts is used [129], exploiting an octree data structure based on a nested dissection of the 3D domain (Fig. 4.15).

Here, a sort of a global stiffness matrix of the system is calculated by assembling the element stiffness matrices, traversing the octree structure from the bottom up (Fig. 4.15). The main advantage here is that when inserting an implant, the stiffness matrices of the cells that experience change are updated locally, and the reassembly step is done only for a modified part of the system. Despite the better overall performance of the solver in comparison to other direct solvers (i. e. Gauss and relatives, where the complexity, as mentioned before is $O(N^3)$), the current reassembling step, which is computationally most expensive, must be interrupted or completely skipped as soon as a user changes the settings.

To sum up, the steps described above are taken within an interactive computing loop, which consists of receiving user updates, pre-processing, solving the aforementioned system of equations, post-processing, and forwarding the results to the user (Fig. 4.16).

4.5.3 Interactive Visualisation

The other component involved in the interactive steering process is a sophisticated visualisation platform that allows the intuitive exploration of the bone geometry and particularly the mechanical response to various load situations in both the physiological and post-operative states of a bone-implant situation in terms of stresses and strains [66, 67].

Settings are updated by, for instance, inserting or moving an implant or testing a new position or magnitude of the forces applied to the bone for each element and corresponding tensor, when a scalar value – i. e. the so-called von-Mises stress norm – can be calculated and visualised, as shown in Fig. 4.17.

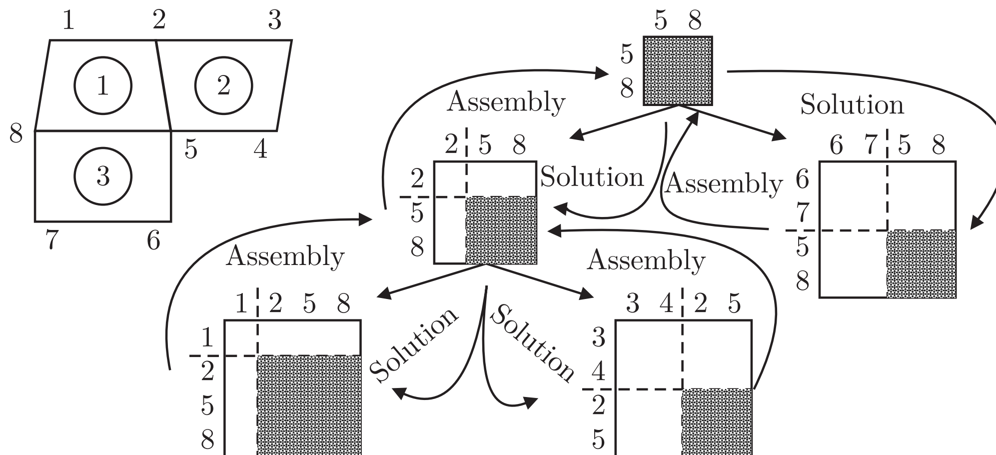


Figure 4.15: ND solver algorithm: Decomposition of the geometry domain results in hierarchically organised systems of equations. First the assembly step is done, where Schur-complements (which result from re-arrangements of matrices separating "local" and "global" DOFs, and partial Gaussian eliminations) are then successively forwarded from the bottom towards the top of the tree. Solution step is done in top-down order and partial solutions are then forwarded successively in the same manner.

4.5.4 Communication

The challenges in developing such a two-component analysis tool are described in more detail in [178, 66, 67, 177, 65]. Unfortunately, due to the rigid TCP/IP socket-based communication pattern between the components in the original version, a new setting can only be considered by the simulation after the result of the previous one has been calculated and sent to the user. In other words, to compute the new result and preview the effect of the latest change, a user would have to also wait for the outdated one. Therefore, the higher polynomial degree used, the longer the total time becomes. Hence, the central topic of this section is also the way in which these two components were coupled in a new approach through the framework to allow for instant feedback to the surgeon. The approach described here along with some of the results presented are also published in [108, 107].

4.5.5 The Framework in Action

The proposed framework was integrated to further improve the tool towards an interactive simulation and visualisation environment. This integration enabled a choice of how many updates to send per time interval, making the environment much more immersive.

The next aim was an instant interruption of a current computation in the case of an update. Due to the update, the stiffness matrices are assembled recursively traversing an octree bottom-up.

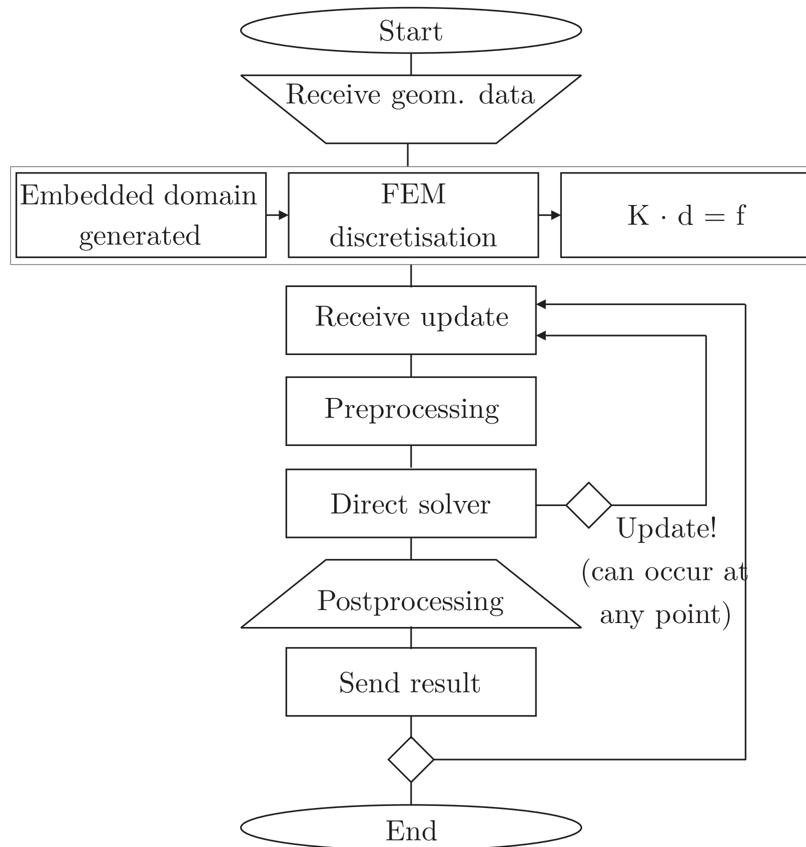


Figure 4.16: Simulation back-end in the Bone application. Interactive computing loop starts with receiving an update, and ends with sending the calculated result to the user.

Afterwards, the system of equations at root level of the octree is solved. The solutions are then always recursively passed to the nodes one level lower in the hierarchy for their own local solutions, as shown in Fig. 4.15. All the partial solutions are finally assembled into the final solution vector. The described algorithm, as presented in [129], shows good scalability values in the case of hybrid parallelisation [130].

The general aim of the framework was to interrupt the most time-consuming phase – i. e., assembly, parallelised using shared or newly developed distributed memory concepts (once they are integrated into the steering environment), or both. This interruption was achieved through cyclically repeated signals checking for updates and restarting instantly if an upcoming message was indicated from the user side. If the update was recognised while processing one of the nodes in the previously mentioned hierarchical data structure, the simulation variables were reset in a way which enforces skipping the rest of the nodes, as shown in Fig. 4.19. All the layers of the recursive assembly function call instantly returned; all the upcoming "solution" steps were skipped as well, and the new data was received at the beginning of the next step of the interactive computing loop.

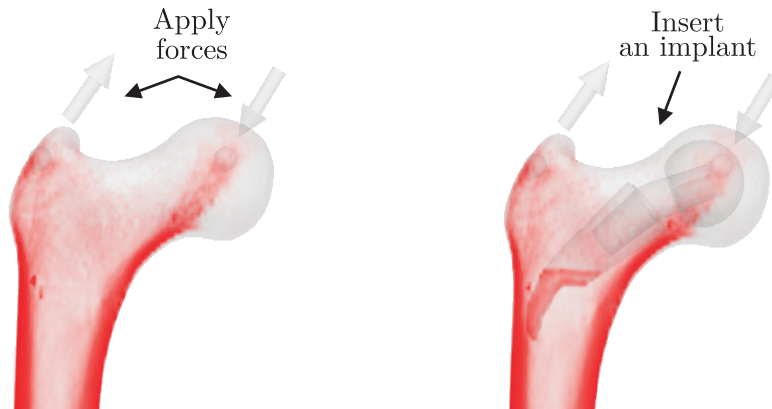


Figure 4.17: User interaction for the *Bone* environment: left – applied forces of a defined magnitude, right – an implant is inserted.

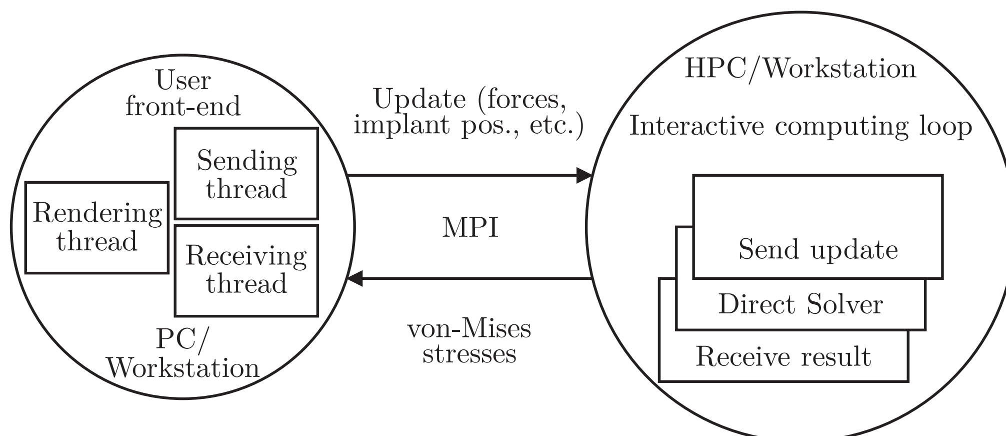


Figure 4.18: The *Bone* setup – on the simulation back-end the so-called von-Mises stresses are calculated and the result is sent to the visualisation front-end for the user to make an informed decision if the implant design/size is an optimal one, or another setting should be provided for the simulation.

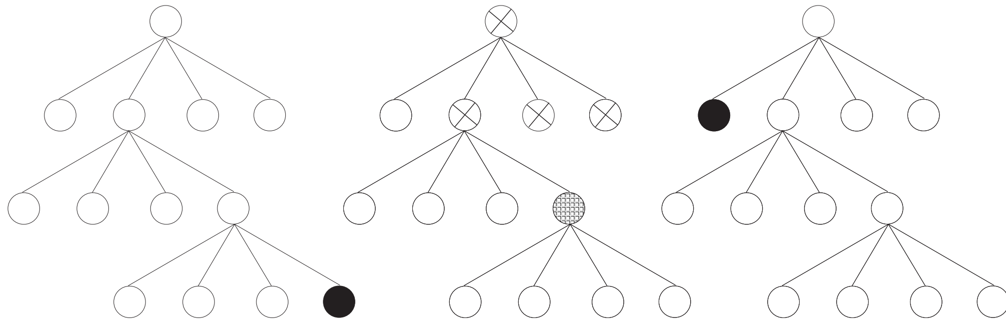


Figure 4.19: Three subsequent steps in processing the tasks (the tree is traversed in the depth-first manner, and the assembly function is recursively called for each existing son-node of the processed node until a leaf node is reached): left – the filled node is the one for which the assembly function is currently called; in the next step (middle) – the interrupt occurs (in the “filled” node), the rest of the nodes are skipped; finally (right), a new computation starts with updated settings.

In addition to the guaranteed consistency data values necessary for accurate program execution, this was an example of where the steps to prevent potentially severe memory leaks had to be taken before the new computation was started. Such leaks could have resulted if the interrupts had happened before the memory allocated in the solver was released. If the solver code was parallelised via OpenMP, it was ensured that when a new update was recognised by the thread catching a signal, all the other threads became automatically aware – via globally shared variables – that they had to skip the rest of the computation.

As soon as the assembly was completed without an interrupt and the solution in terms of stresses calculated, this result was sent back to the user process for a visual update.

Although a significant amount of time was saved by calculating results only for an actual setting and skipping all previous ones, for p values higher than 4, a delay was noticed, as expected, since the time needed for a new computation increases when p is increased. Here once again, a hierarchical approach was beneficial.

4.5.6 Hierarchical Approach

The hierarchical approach used in this test case was based on using several different polynomial degrees, with a simulation for each running as a separate MPI process (Fig. 4.20).

The voxel data as well as the data referring to user interaction was sent to all of the processes via MPI, allowing them to start their own computation. Computations for a lower p were finished faster than for a higher p. As soon as the first was finished, the results were sent to the front-end to be visualised. Then the results for a higher p followed. Alternatively, one might have used adaptive p-

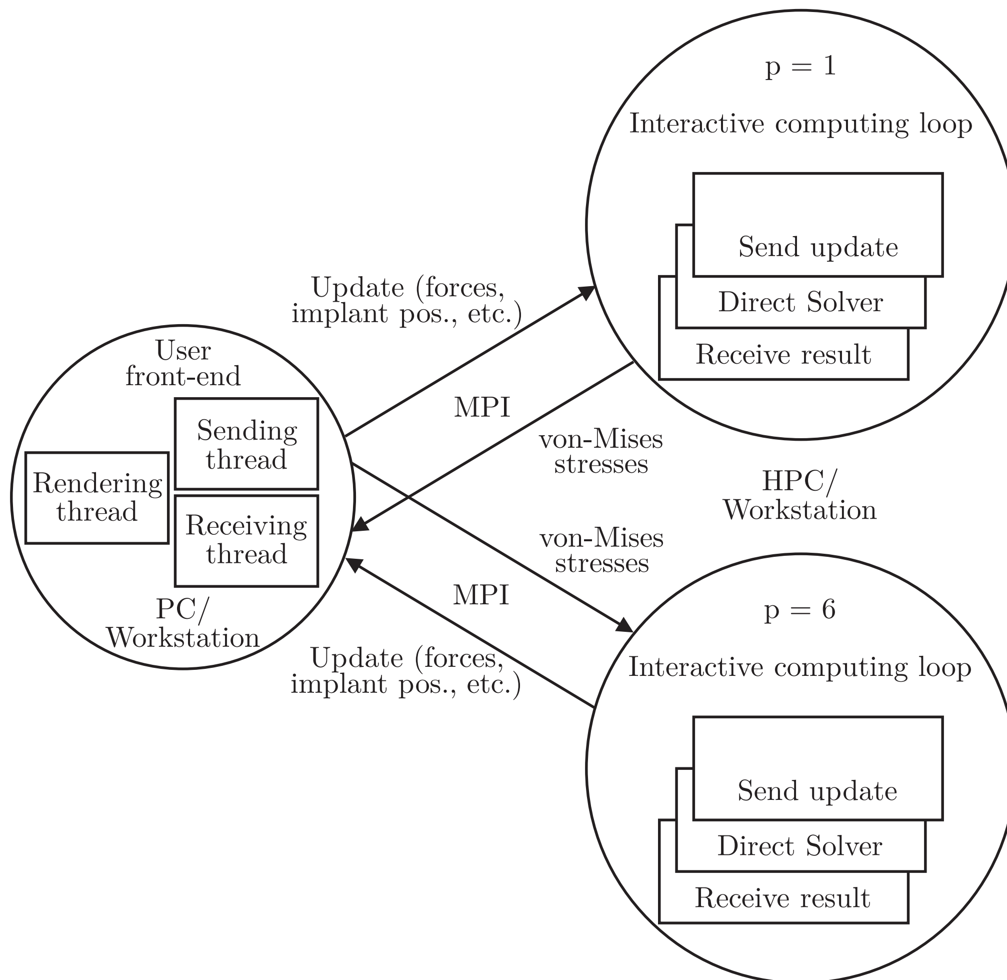


Figure 4.20: Bone: hierarchical approach – on the simulation back-end, several MPI processes are run for different polynomial degrees (p). While a user is interacting, due to the frequent interrupts, only the result for a lower p is sent to the user, while as soon as he stops, the result for a higher p has enough time to be calculated and sent.

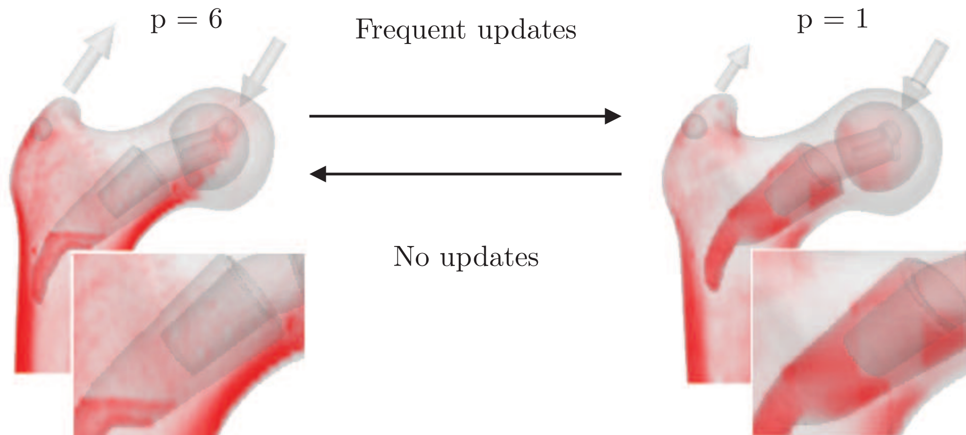


Figure 4.21: Switching from the lower to the higher polynomial degree (achieving a higher accuracy as can be observed in the picture), and vice versa, depending on the occurrence of user interaction.

FEM approximation as proposed in [144] in the context of computational steering and briefly described in Subsection 3.4.3). With this approach, the integration of the framework with support for a hierarchical approach would have been even more straightforward. In that case there would have been no need to run several instances of the simulation program and communicate data to and from each of them. With the scheme developed here, however, the same effect was achieved, without investing significantly more time in implementing a hierarchical approximation method.

What is accomplished is that while the user's interplay with the settings is very intensive, he gets immediate feedback about the effects of his changes, namely results for lower p (e. g., $p = 1$ or $p = 2$). He can see the more accurate results in addition to this only as soon as he stops interacting and the simulation has a chance – now without enforced restarts – to finish one iteration in the interactive computing loop for higher p values (Fig. 4.20).

As soon as user interaction starts over again, the results for the lowest p are immediately calculated again and visualised; hence, the procedure described above is repeated. The user can choose the number of MPI program instances executed for different p values (i. e. hierarchy). In addition, every opportunity must be taken to accelerate the computation for all the p values used.

Thus, the next subsection discusses tests with different parallel versions of this particular simulation. Employing a parallelisation strategy should not be seen as the responsibility of the framework; however, it is an important topic to discuss in the context of an optimal simulation basis. A parallelisation strategy within an interactive environment in particular should not only give good speedup results but also not cause any significant overhead to each interactive computing iteration. The following scenarios thus show one logical thread of progression

which seeks a good trade-off between the two aforementioned conflicting goals and results, on the way, in various interesting conclusions.

The non-optimal speedup achieved in the simplest, multithreaded scenario was especially noticeable while increasing the number of the computational resources. To remedy this, research was also done in the direction of message-passing-based parallelisation for the ND solver. The solver itself is most time consuming within the interactive computing loop, thus, a stand-alone version was used to explore the parallelisation possibilities, with the intention to integrate it later into the existing complex environment. Load balancing techniques that are as efficient but also as simple as possible were applied in order to achieve a fair but fast distribution of the tasks among the processors throughout the interactive process.

This turned out to be challenging for the tasks organised in the hierarchy, where the number of processes involved typically decreases by the factor of 2^n on each level, where n is the dimension of the space. Thus, an optimisation technique for the tasks with dependencies was applied, involving various heuristics in order to involve all the available processes during the overall program run time. This produced promising preliminary results. All of these points are thoroughly discussed in the subsection to follow.

4.5.7 Parallelisation for Hierarchically Organised Tasks

Problems involving spatial decomposition techniques [129] based on Schur complements may result in hierarchies of tasks with bottom-up dependencies, such as trees, as described in Chapter 2. This can be very advantageous in an interactive steering environment, because an update of one part of the domain leads to the re-computation only for that particular sub-tree [130]. Parallelisation in the case of balanced trees – resulting from bulky structures, such as cubic-shaped structures – typically results in a straightforward load-balancing strategy largely based on data locality. In other words, tasks belonging to a certain sub-tree are mapped to a certain processor, which easily achieves a fair work distribution and minimises communication costs. In the long structures such as a femur, the situation is not so simple, especially if one considers the aforementioned prerequisite that the load balancing step, which has to repeat on every update, therefore must be as computationally cheap as possible to not slow down the overall interactive process.

Multithreading in ND solver

The first, simplest parallelisation strategy applied for the ND solver was multithreading, in which the tree of tasks was parallelised on the first level below the root, where up to eight child nodes resided, and the corresponding sub-trees could be assigned to different threads for concurrent execution. Fig. 4.22 shows this multithreaded strategy's speedup results – for the most consuming assembly function in particular – which resulted from the tree being unbalanced.

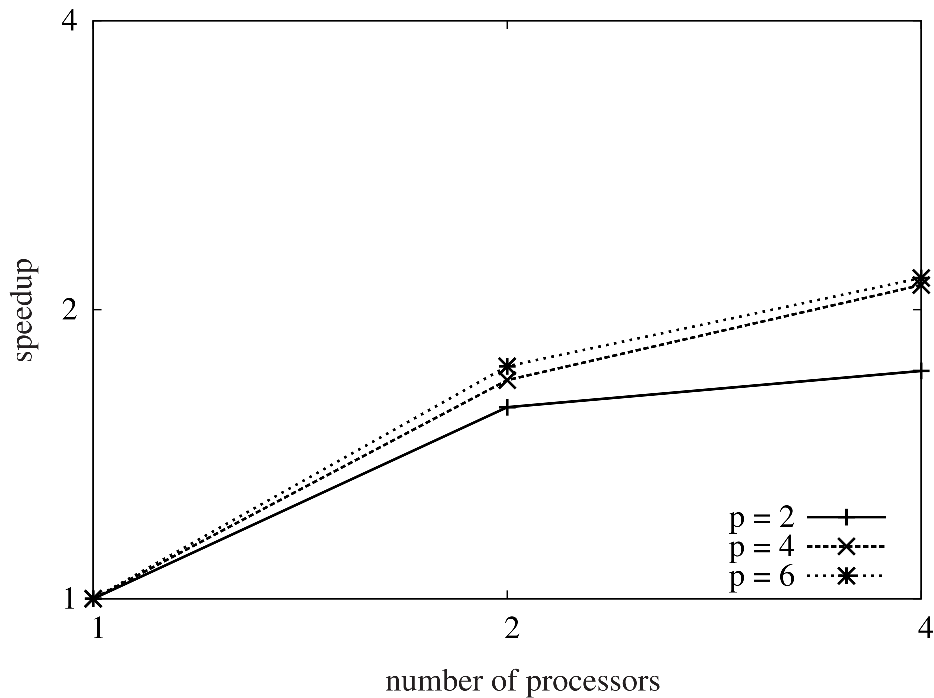


Figure 4.22: Bone multithreaded version of ND solver – the one integrated currently in the interactive computing environment – speedup results.

A reader may also find the results for the execution times from Tab. 4.1 published in [179]. The article shows a comparison of the different resolutions $1 \times 1 \times 1$ and $2 \times 2 \times 2$ (with half of the initial resolution) and of different computations cycles on an eight-core machine (initialisation vs. steering time – 1 iteration), as well as some additional information about the simulation.

Fig. 4.23 shows the speedup results based on the information in [179], both for the solver of the system of the equations – where the assembly function is the most time consuming – and for the overall iteration in the computational steering loop. These results are similar to those in Fig. 4.22, where only the assembly function is taken into account.

Another simple multithreading parallelisation strategy was experimented with in a stand-alone ND solver, in order to draw more conclusions. This second strategy is based on pure data decomposition in the calculation of Schur complements (partial Gaussian elimination algorithm). The speedup results for static scheduling are shown in Fig. 4.24.

It is noteworthy that in this case for $p = 4$ and $p = 6$ the speedup *decreases* with the increase of the polynomial degree p (i. e. the problem size in this static scheduling approach). This is due to the dominating partial Gaussian elimination algorithm within the assembly function, where, using this strategy, not all the threads get a fair portion of calculation. To improve this, one may experiment

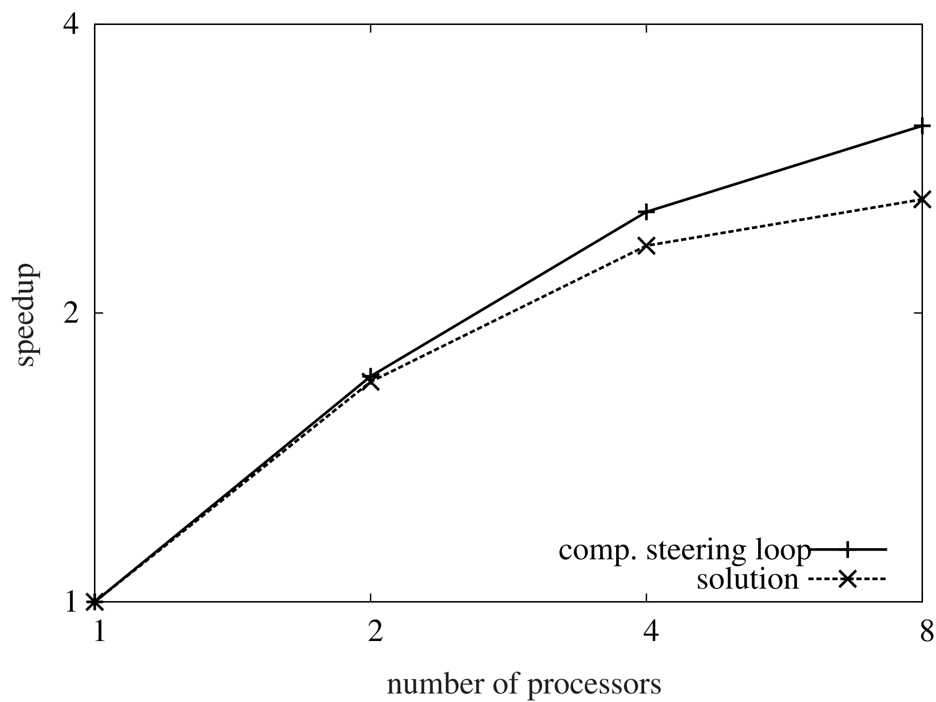


Figure 4.23: Bone multithreaded version, the graphs are based on information presented in [179] – measurements on 8 core Dell Precision T5500, Intel Xeon W5590 CPU 3,33 GHz, for $p = 4$: measurements of the speedup of assembly function, vs. the speedup of the whole interactive computing loop have similar results.

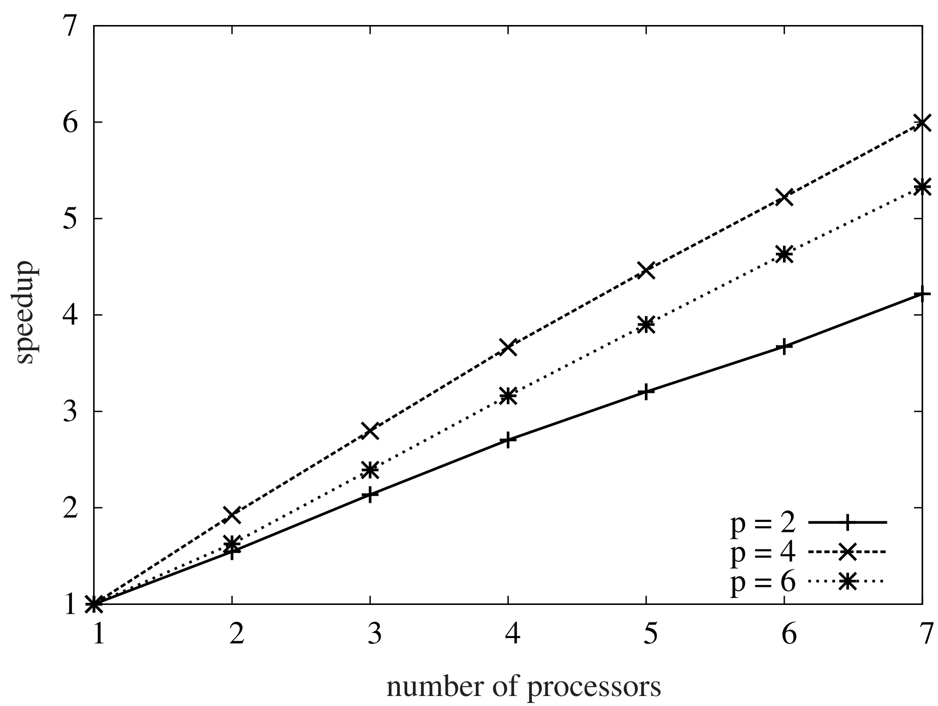


Figure 4.24: Bone multithreaded – an experiment based on the data decomposition on the level of the second nested loop in the partial Gaussian elimination – speedup results on eight Intel(R) Xeon(R) CPU W5590 3.33GHz.

Model		$1 \times 1 \times 1$	$2 \times 2 \times 2$
DOF		61,368	13,698
Initialisation time[s]	system setup	4.73	2.50
	system matrices	0.78	0.16
	Total	5.51	2.66
Steering loop	Total	8.57	0.75

Table 4.1: Comparison of the execution times for one iteration of the steering loop and for different execution cycles with $1 \times 1 \times 1$ and $2 \times 2 \times 2$ (2 times smaller in each spatial direction) discretisation resolutions. The measurements are performed on 8-core Dell Precision T5500, Intel Xeon W5590 CPU 3,33 GHz.

with different scheduling techniques, such as using dynamic chunk scheduling and different chunk sizes.

Another remarkable point is that despite the obviously better speedup for the second scenario, the execution times for those two multithreading strategies are incomparable, because in the first one (with worse speedup) the highly optimised "LAPAC package is used" [179] within the solver; thus, it performs better for up to 8 cores *in terms of the execution time.*

The observed results have, however, led more easily to the conclusions when designing and testing an optimal distributed-memory-based parallelisation, where more computing resources can be exploited. A long-term vision is to achieve the update rates in the interactive environment that fall even below one second for polynomial degrees higher than 4.

Distributed parallel ND solver

A good scheduling strategy for an interactive computing environment, as mentioned before, typically involves a trade-off between achieving an even work load distribution among all the processors (i. e., a balanced assignation of tasks to the computing resources) and keeping both the communication and optimisation costs low. Finding an adequate trade-off here should result in a good speedup of a parallel application, even if the re-scheduling step has to be done every time a user interacts.

Like in the multithreading case, for hierarchically organised tasks with bottom-up dependencies, such as in the octree structure generated for a femur, the "classical" scheduling would be based on the task locality (i. e. the property of belonging to the particular sub-tree). However, the resulting tree for long structures, such as a femur, is not well balanced. Moreover, the number of processors which can be simultaneously exploited decreases by a factor of eight at each subsequent level approaching the root of the tree.

Dynamic load-balancing strategies using hybrid parallelisation patterns perform well for the nested dissection solver [130, 105]. However, in interactive applications – which assume frequent updates from the user's side, frequent changes in the state of simulation and tasks, as well as in the computational model itself – are not favourable. Namely, user updates result in extreme dynamicity of the system itself and thus an overhead-prone load-balancing step must follow. Static load-balancing strategies are therefore estimated as more favourable in this case.

What is needed, consequently, is a scheduling optimisation approach that is both effective yet simple and efficient, and that does not obstruct the desired interactive process. To the author's best knowledge, such a solution is not the present focus of the existing sophisticated optimisation strategies [109].

A "classical" approach

From the first, simplest, multithreading scenario, it is clear that splitting the tree of tasks among the processors by cutting the tree on the first level would not lead to satisfactory speedup results for the femur model in consideration.

A good dynamic load-balancing strategy, where the tree is cut on different levels (see Fig. 4.25, left), is proposed in [101] for a hybrid parallel scenario (involving OpenMP and MPI). However, this load balancing strategy is estimated to be too computationally expensive, and thus, an attempt has been made with a simpler approach.

In Fig. 4.25 (right) the basic idea is illustrated: first, the tree was cut at a certain level depending on the number of available processes – i. e. at the first level for 2^N processes, where N is the spatial dimension, at the second for 2^{2N}, etc. In the same figure, a binary tree ($N = 1$) is represented for simplicity's sake instead of an octree ($N = 3$), which is what is actually used in the Bone application. The corresponding sub-trees were then assembled concurrently. Only one of the possible processors took over in the upper, unassembled part of the tree – i. e., in the ancestor nodes of those on the last-assembled level. Namely, one of the processors involved in the computation of the son nodes did the computation for a parent; this was done successively by level.

A low speedup is shown in Fig. 4.26 due to the decrease in number of processors that can be employed at each level moving up the tree and due to the MPI internal decisions related to rendezvous protocol, which are explained in the subsections to follow. The latter effect might be improved by receiving all the available data as soon as it has been sent. Moreover, further research would be useful in refining the task granularity when reaching the root node of the tree.

A possible limitation of this approach may be that the number of exploitable processors is still limited by the number of leaf tasks. Another scheduling strategy that is even better optimised for the case of extremely unbalanced trees will now be described. Until now, this new strategy has been evaluated only for the numbers of processors currently available on our clusters, as published in [109]; however, first results are promising and the implementation has led to better

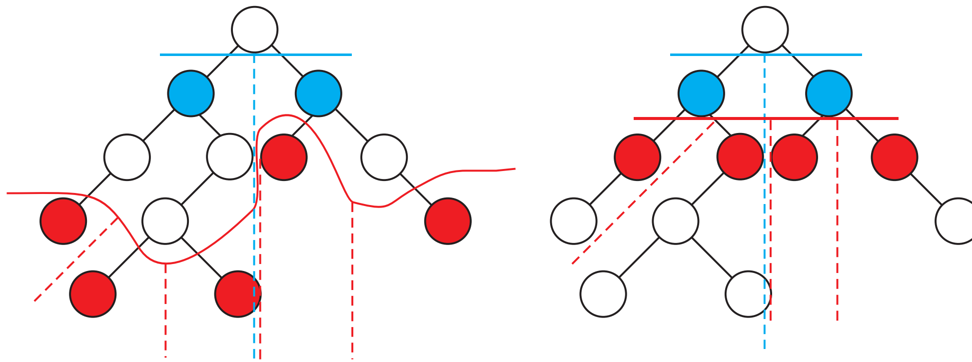


Figure 4.25: Bone distributed parallel version (a binary tree is presented for the reasons of simplicity, although an octree would be appropriate): left – a sophisticated dynamic load balancing technique, too computationally expensive for interactive computing; right – a static load balancing technique, proposed for interactive computing environment.

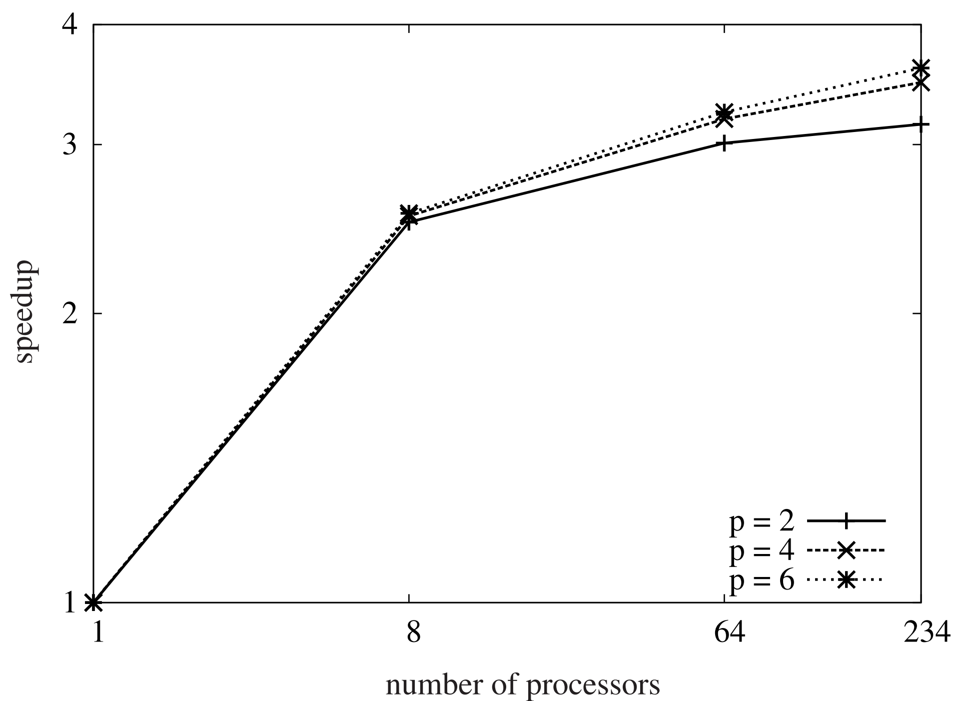


Figure 4.26: Bone distributed parallel version – an initial static load balancing technique, proposed for interactive computing environment – speedup results for $p = 2$, $p = 4$, and $p = 6$ on Shaheen IBM Blue Gene/P supercomputer owned and operated by King Abdullah University of Science and Technology (KAUST). The maximal number of processors for which speedup is measured is 234, since this is for this particular data input the number of tasks in the leaf nodes of the tree, i. e. the maximal number of the tasks which can be processed in parallel at any point.

understanding of some issues with the "classical" approach in relation to MPI internal decisions (see 4.5.7).

Optimisation

The newly proposed scheduling strategy is described in [109]. What was assumed to be known a priori for a particular input data are the sizes of the tasks resulting from the initial task decomposition of the problem, data dependencies and synchronisation requirements. Traversing the tree bottom-up, one had to estimate the number of operations (partial Gaussian elimination operations) needed for processing at each node. This was considered the *weight* of a particular node – i. e., a rough estimation of the amount of work to be done in the nested dissection solver for the node. Common assumptions were not made, "which are apt to have restricted applicability in real environments" [116], such as the target parallel architecture or uniform task execution times.

The sizes of the tasks vary notably. In order to avoid single processors becoming bottlenecks, due to the considerably large tasks they might need to process by themselves, a single task was split among several processors when mapping all tasks to the processors. This splitting was done based on the comparison of a task's estimated work with a "unit" task, which refers to an octree leaf or to a single element in terms of FEM [109]. However, unlike in the "classical" approach, the data locality was neglected in this scenario, and the pure load balance is highly prioritised in this stage of the development.

Since the scheduling problem can be solved by a polynomial-depth backtrack search – and is thus NP complete for most of its variants – efficient heuristics must be devised [109]. When making a strategy, one must take into account the two major factors: the sizes of the tasks and the dependencies among them. Therefore, at least two aspects have to be considered. The first is the level of the task dependency in the tree hierarchy (i. e., children nodes have to be processed before their parent nodes). And second, if the tasks are of the same *dependency level*, the distinction is still made between their different levels in the tree hierarchy. This property is named the *processing order* (Fig. 4.27). Assuming the depth of the tree is N, the tasks from level M in the hierarchy have the processing order of $N - M - 1$ [109].

A list of priorities was formed by searching for the tasks with both the lowest level of dependency and the lowest order of processing. Consequently, what was prioritised were the tasks belonging to longer branches of the tree structure, with an estimated bigger overall load. Following state-of-art heuristics [87], a so-called *max-min* order was given to the tasks with exactly the same priority according to the previous classification if there were more tasks than available processors in the same priority list. This way, big tasks – in terms of their estimated quantity of computations they involved – were the first to be assigned to the processors. The main principle of max-min order *was* followed; however, only partial sorting was applied in order to avoid additional computational complexity of sorting of the tasks. Namely, the priority lists were filled by adding tasks bigger than a

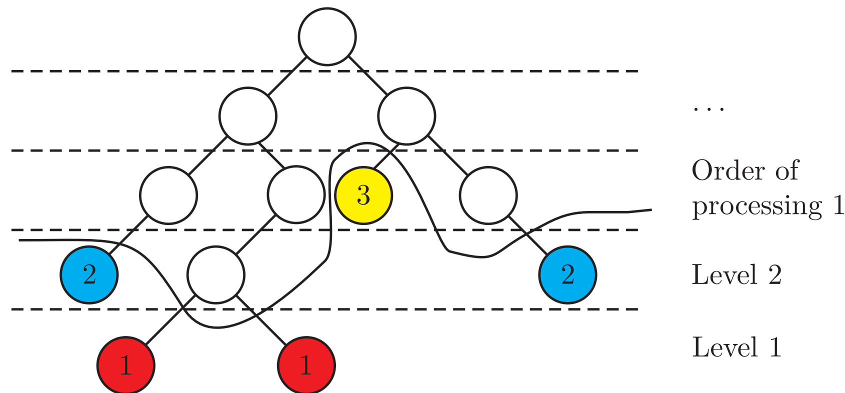


Figure 4.27: Dependency levels (division by horizontal lines), and processing order (numbers within the nodes).

certain predetermined limit to the front of the list, and those that were smaller to the end.

This procedure formed arrays of tasks – so-called “phases” – each of which consisted of as many tasks or their parts as there were computing resources [109]. Tasks taken from the priority lists were assigned to the phases in the *round-robin* fashion. As mentioned before, tasks were also split among several processors if needed. The *phases* correspond to the mapping to be done. The “bulky” tasks are those whose size divided by the number of available resources exceeds the size of the smallest processing unit; these tasks were simply split equally among all the processors. The results in Fig. 4.31 illustrate that the capacity of each phase was mostly as “full” as possible; in other words, all the processors were busy with approximately equal amounts of work throughout the solver execution. This was exactly the intended effect.

Communication pattern

The mapping concept was a typical master-slave setup. Initially, all the slave processes received the tasks they were in charge of and started processing them sequentially. The tasks without dependencies could, clearly, be processed immediately. Others needed information from other processors. Thus, one of the following scenarios took place: (1) for the tasks that could be instantly processed, the assembly step was done and the resulting data was sent to the processors needing exactly this data for the assembly of their own tasks; (2) for the tasks dependent on other tasks, all the related data had to be received before the assembly. In the present implementation, both the data necessary for a certain step and all the pending data needed later was received, in order to reduce the latencies caused by the MPI internal default decisions [109]. This is elaborated upon in the next subsection.

When processing (a part) of the task was finished, all the processors with a part of the same “bulky” task were supposed to exchange data amongst themselves.

Finally, only the first processor – from the list of processors in charge of the task – sent the result to those that were not in charge of the same task, if they would need this data later. The pattern was repeated until the root node was reached, when, due to the node size, all the available computing resources often become involved [109].

Improvement for distributed scenario

An interesting issue, based on the information provided in [80], is that internally in MPI a process P_1 may possibly send a message before P_2 is ready to receive it. The receiving process, thus, has to remember that a message has arrived and it must store the data somewhere. To remember that a message has arrived but has not matched a receive – a queue of messages that were "unexpected" is kept. When a program tries to receive a message with MPI_Recv, it first checks this "unexpected queue" to see if the message has already arrived. If it has, the receive can remove the message and the data from the queue and complete. The receiving process must also store the data somewhere. And even if data is too big to fit the available memory or buffer, MPI standard requires the implementation to handle this case and not to fail. There are ways of course to influence the internal decisions of MPI, the increase in the receive buffer, the control of "rendezvous protocol" activation, etc. But such measures typically downgrade the performance; thus, it is better to modify one's own code.

Based on the Vampir (Visualization and Analysis of MPI Resources [16]) analysis, and led by the information from [80], an improvement in the described distributed scenario was made in the way that all messages available for receiving were instantly received. The comparison of the results is presented in Fig. 4.28 and Fig. 4.29 – before and after the improvement, respectively. In the visual output from the Vampir [16] analysis tool, the original colours have been replaced and a few elements added to improve the visibility of the desired features for this text.

4.5.8 Results

The results refer to the overhead of the framework within the interactive environment (multithreaded scenario) as well as to the preliminary speedup results and an evaluation of the scheduling optimisation for the distributed parallel scenario (which is not yet a part of the interactive environment where the framework is integrated).

Overhead of the framework

In this particular test scenario, the simulation was executed on multi-core architectures and connected to the visualisation front-end via a network. Evaluation of the performance still proves that this is yet another test case where the overhead caused by the framework itself is not significant [108, 107] (see Fig. 4.30). The intuitive interaction from a user's point of view also runs "smoothly".

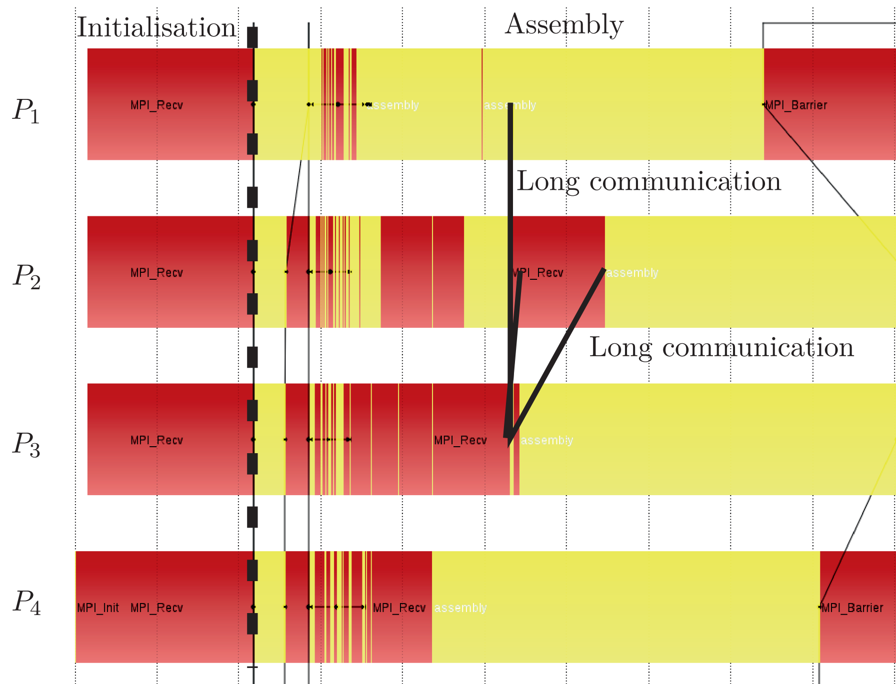


Figure 4.28: Vampir results for an "asynchronous" run: distributed parallel assembly function of a "stand-alone" ND solver run for 4 processors. The dark (red) color represents the time spent for communication calls and light (yellow) color – the time spent for doing the intended computation. The marked thick long lines from P_3 to P_1 and P_2, e. g. represent the long communication intervals from the point when the message is sent until the point when it is received. This, however, takes unexpectedly long due to the MPI internal decisions. The execution time takes, based on several tests, a few seconds longer than expected for each processor.

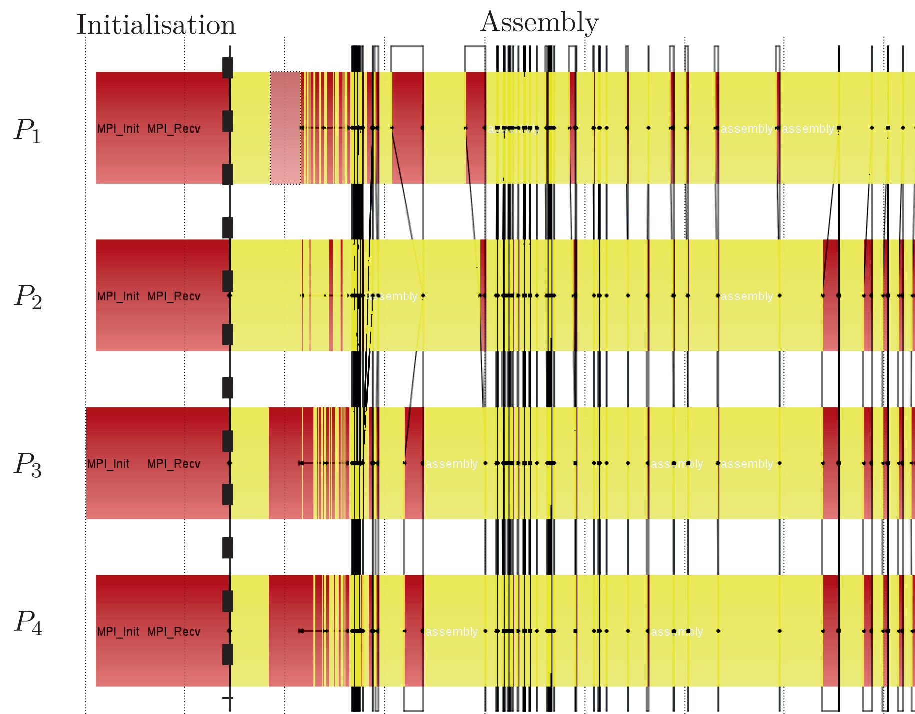


Figure 4.29: Vampir results for a "synchronous" run: the same simulation, where all the data is received as soon as possible (greedy algorithm). The dark (red) color represents the time spent for communication calls and the light (yellow) – time for doing intended computation. The improvement of the total communication time, based on several experiments, is around 30% and of the execution time is around 15%.

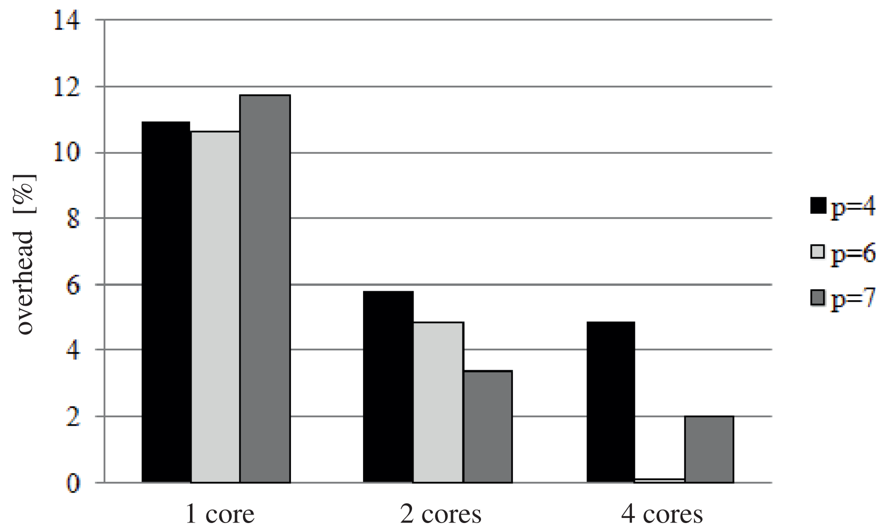


Figure 4.30: Bone multithreaded – the version currently integrated in the interactive computing environment – the overhead of the framework. Along the horizontal axis the number of cores is represented and the vertical axis shows values in percentage for $p \in \{4, 6, 7\}$, respectively.

Scheduling optimisation

The distribution of the tasks themselves can be observed from Fig. 4.31. This distribution results from the task priorities and the capacity of so-called *phases*. The capacity of a phase is equal to number of available processors. The aim is to completely fill all the phases, keeping all the processors busy throughout the simulation execution time.

Speedup results

The speedup results for the described message-passing-based parallel scenario, with optimised scheduling strategy, are shown in Fig. 4.32 and Fig. 4.33. The satisfactory speedup is observed for up to 16 processors for different polynomial degrees of the basis functions in the Finite Element approximation, where higher polynomial degrees correspond to larger problem sizes. Compared to the load imbalance effects in the second presented multithreaded scenario, where the data decomposition is done in a similar way, here the speedup curves improve with the increase of the polynomial degree. This is due to the *round-robin* mapping strategy, which, in this case, does not always assign larger portions of work to the same processors. Results for a larger number of distributed memory computational resources should also be tested. The expected speedup is achieved for up to 16 processes on the hardware available at our department; however, with high-bandwidth connection between all cores, the trend would be similar also for larger number of processors (see Fig. 4.32 and Fig. 4.33). According to this tendency observed for $p \leq 6$, engagement of a larger number of processors would result even for $p \geq 6$ in the desired rate of at least several updates per second

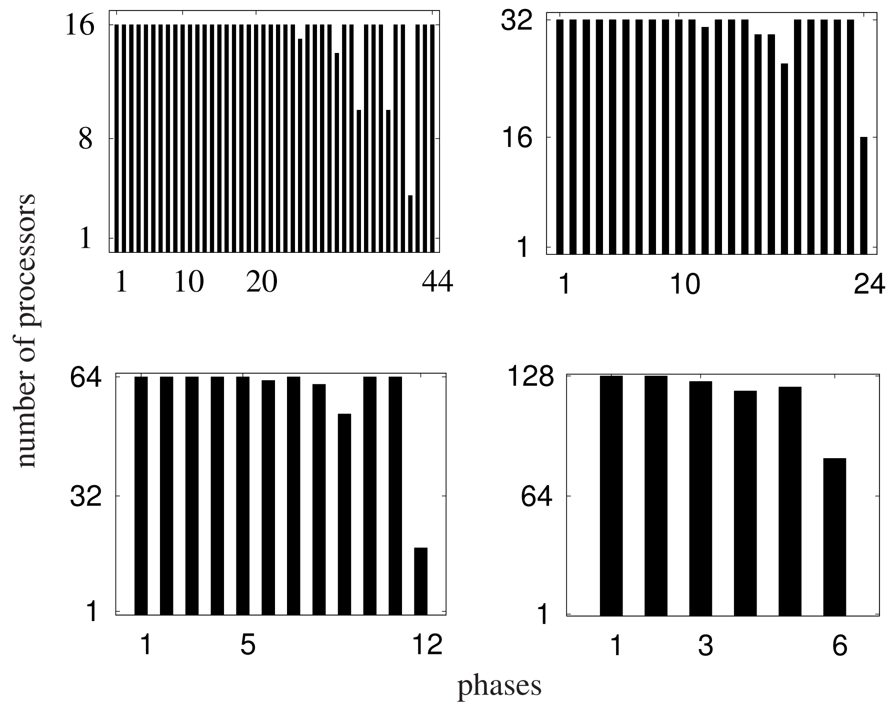


Figure 4.31: Phases in the top-down, left-right order, respectively for $N \in \{16, 32, 64, 128\}$ processors available. Horizontal axis – the *phase number*, where one phase contains the (up to N) tasks which have all the dependencies from other tasks resolved, i. e. can already be executed; vertical axis – the capacity used, i. e. the number of processors (and the corresponding tasks) involved (out of the maximal N).

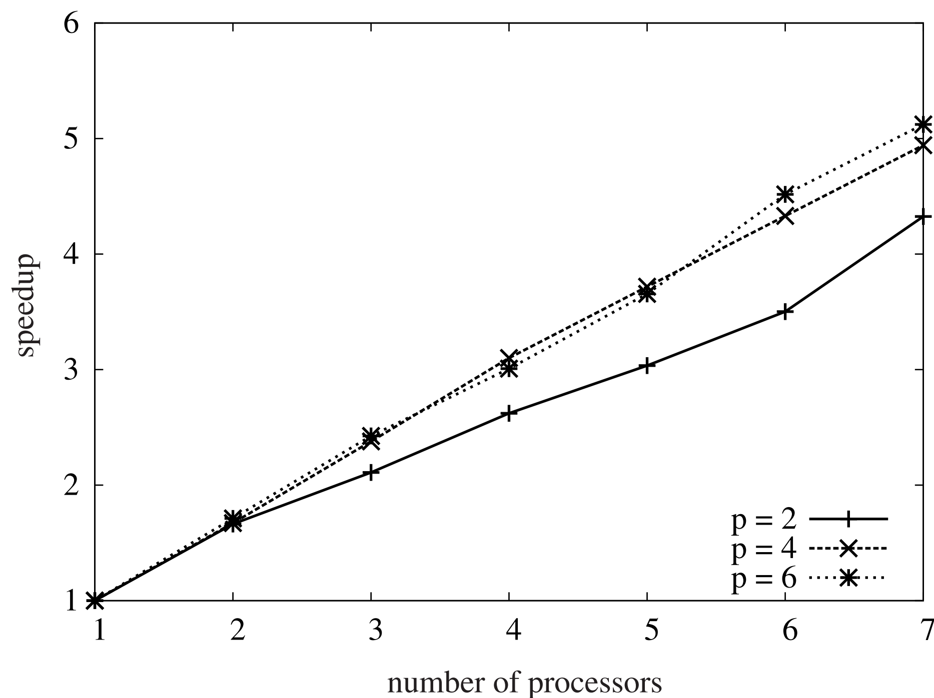


Figure 4.32: Nested dissection ($p \in \{2, 4, 6\}$) distributed parallel solver (a “stand-alone” version) with optimised scheduling – promising speedup results for a small number of processors – Intel(R) Xeon(R) CPU W5590 3.33GHz.

for the calculated bone stresses in our interactive environment while the user modifies the magnitude of the applied forces or the position of an implant [109].

4.5.9 Effort to Integrate the Framework

The following steps had to be taken to integrate the framework. They refer back to the existing interactive computing environment, without the integration of the developed distributed parallel solver to speed up the computations (i. e., the version which supports multithreading).

- The framework function for overriding the default SIGALRM signal handler was inserted into the simulation code and SIGALARM set to occur in a predefined interval.
- The loop variables manipulated within this function to skip the redundant computation were declared global, *sig_atomic_t* and *volatile*.
- The signal-handler function was edited according to the needs of a specific code and compiled together with the rest of the code.
- The implementation of the signal handler function in this particular case consisted of refreshing the signal action. This was necessary for the Intel

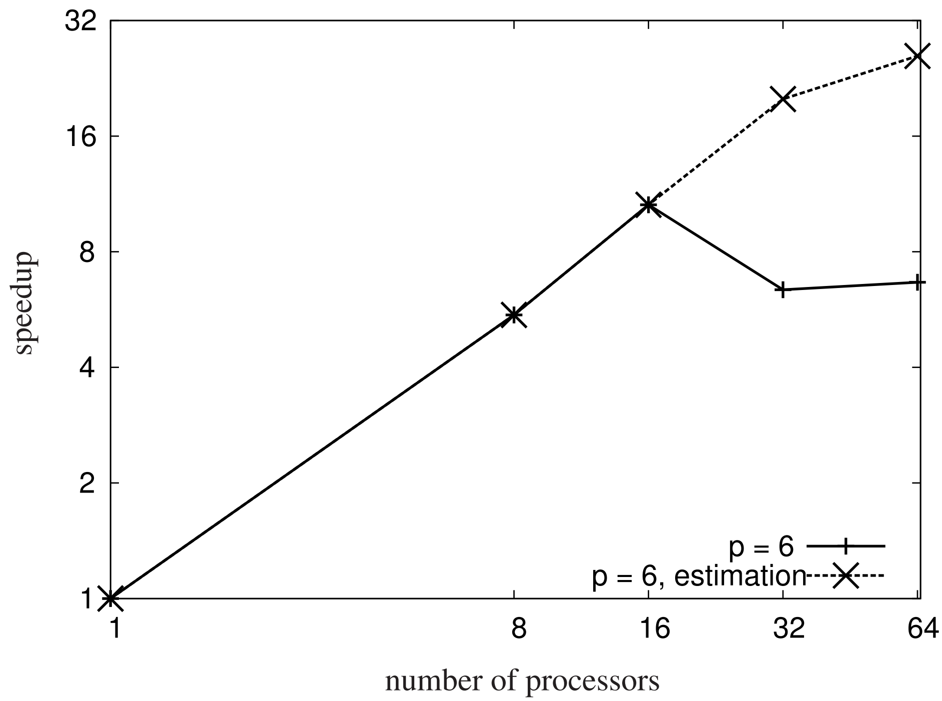


Figure 4.33: Nested dissection ($p = 6$) distributed parallel solver (a "stand-alone" version) with optimised scheduling – measurements at *sandstorm* cluster (4 nodes – each 2 physical Intel Xeon E5-2690 8 core processors running with 2.9 GHz– connected via 1G Ethernet switch). Each node has 16 cores with high-bandwidth connection, which dramatically affects the speedup results for more than 16 processors. The estimation of the result is provided (based purely on the network bandwidth information) in the case of the same high-bandwidth connection between all the nodes for 32 and 64 processes. If the network latency would be taken into account, the estimation result would be even more promising.

compiler, although the system conforms to POSIX standard and provides *sigaction* functionality. The signal handler function probed the message from the user and, if needed, manipulated a specific variable involved in the computation within the assembly function of each octree node. This way, the processing of the rest of the nodes was skipped.

- A straightforward reinitialisation step for the manipulated variable had to be done at the beginning of the interactive computing loop.
- Communication based on a rigid pattern was replaced with the corresponding framework function calls.
- To enable a hierarchical approach, several instances of the program were compiled, run, and made accessible by the framework central entity, communicating the data to and from the computations for different polynomial degrees. Just as in the previous application cases, this was not considered the basic feature, nor the responsibility of the framework; however, it was implemented within this work.

The greatest challenge, taking several working days, was creating the new communication pattern, especially to support the hierarchical approach. The functionality for interrupting the computation to check for updates and restart a computation if needed was quick and straightforward as usual. However, the unexpected challenge was that the default action for an occurring signal had to be refreshed every time a signal was caught (via *sigaction*) in order to be effective for the version of Intel compiler used. This is normally not the case on the systems supporting *sigaction*, which generally has major advantage in this regard over *signal*, for example.

4.6 Conclusions

This chapter described the four application cases in which the framework was tested, as well as the integration process itself. Moreover, it presented the results in terms of the overhead of the framework and the user effort to integrate it for all the diverse test applications and for different problem sizes.

The 2D heat conduction simulation was the first, simple test scenario, where the user component and the C++ simulation program communicated data over a network. The simulation itself is a sequential code, where a hierarchical approach was applied using multiple grids for FD approximation. The coarsest level was used while the user was interacting, and then an upsampling procedure was applied to accelerate the convergence on finer grids once the interaction became less frequent. The framework performs well in terms of the overhead and is also intuitive from the user's point of view.

As concluded in [104], interactive computing represents a new feature in the *AGENT* code and is – according to scientists at the UNEP – the first scenario in neutron transport codes based on deterministic methods where resolution

parameters are adapted "on-the-fly" to accelerate convergence. Although it is in its early stage of development, the integration of the framework provides a number of advanced ways of running *AGENT*, allowing a user to monitor the solution for the best estimate in a shorter computation time during the reactor core modelling [104]. For example, by starting the *AGENT* solution with a coarse resolution and interrupting the solution in an early stage of computation, users can accelerate the overall time to convergence. Also, during run time the user can correct initial mesh and resolution parameters, increasing the accuracy of the entire simulation. Moreover, interactive steering is well utilised as a tool to validate and verify the code simulation. The next steps at the UNEP would be to enable the user to steer geometry mesh parameters and monitor "live" the graphical representations of the main solution parameters such as neutron flux and current distributions in 2D and 3D solution modalities. From the framework implementation point of view, *AGENT* is the first Fortran code where it has been integrated. Valuable experience has been gained in this study for supporting such codes in the future.

Significant advantages of the *SCIRun* software package over other test applications turned out to be its modularity and the fact that it is based on a dataflow model. Due to its modularity, the best interfaces to the framework were easily recognised. The dataflow model contributed to the automatic re-execution of only necessary modules. The underlying Model-View-Controller design pattern, however, has introduced a few issues related to making the Controller entity cancel scheduled, but outdated, jobs. The three simulation scenarios were tested in order to estimate the overhead of the framework. Tests were made for different update intervals (5, 2, or 1 millisecond), for different solver methods for systems of linear equations (with or without pre-conditioning) and solver-related parameters in *h*-FEM approximation. The overhead of the framework is up to 16% of the execution time. Even in the case of this longest potential overhead, other alarm intervals can be chosen to keep the overhead below 10%. This result shows that it is worth experimenting with different alarm intervals for a specific simulation in the case that the execution time has been significantly extended. The steering process itself now runs intuitively and smoothly. For all the data sizes tested the immediate visualisation of the result as a response to user changes is made possible within a second. However, additional re-use of previous computation results for new computations should be considered when changing certain parameters (if applicable for some of the scenarios). It is also possible to test the integrated framework for any other simulation cases without additional code changes. From the user effort point of view, the integration of the interactive computing framework has turned out to be quick and straightforward, quite similar to the other application scenarios [103, 107].

The starting point of the work related to the *Bone* application case was a computationally efficient simulation and a sophisticated user interface with visualisation module. Both components were with their features opening the door for real-time interactive computing. With the integration of the framework, this environment has not only become more suitable for this purpose in the way the data is com-

municated, but it also enabled the simulation to be interrupted as soon as, and every time, the user interacts. As a result, instant feedback from the simulation is gained as a response to user intervention.

In addition to the basic framework functionality, which was straightforward to integrate, a hierarchical approach was implemented, inspired by p-FEM natural hierarchy. A static load-balancing strategy for the time-consuming ND solver was estimated as optimal for the interactive distributed environment. As a stand-alone version, the strategy promises to effectively exploit the available processing units throughout the execution of the simulation program, although it has yet to be tested and integrated within the environment. The speedup results achieved so far are very promising; however, testing it for a larger number of processes will show whether the desired update rate for the stresses of the bone under the load is going to be gained, even in the case of intensive user interaction. In future work, reduction of communication times by also taking into account the data locality may be considered. Such an approach must attempt to find a balance between its benefits and inevitable computational costs.

4.6.1 Hierarchical Approaches

It has been shown that the hierarchical approaches the framework itself supports are not limited to, for example, recursive coarsening of the grid when the user interacts and then upsampling when the interaction stops, as in the first, simplest 2D heat conduction simulation scenario. On the contrary, other simulation-specific hierarchies are equally supported and encouraged when applicable for most productive symbiosis with the framework. For example, different polynomial degrees of basis functions in p-FEM scheme approximation may be used, as well as different numbers of azimuthal angles for MOC simulation. Any user of the framework can, if needed, adopt “the hierarchies” to his individual requirements. One potentially valuable idea is to start separate simulation processes for different “hierarchies” for FD or h-FEM – as is currently done in the *Bone* project.

4.6.2 Effort to Integrate the Framework

In sum, in the shared memory simulation scenario, the only interfaces in the framework are the data steered by a user, variables that are manipulated to restart the computation and a couple of standard function calls overriding the default signal behaviour. This shows that the approach is as minimally invasive, comprehensive and straightforward as possible. However, one must note that these variables to be steered are globally visible in the code (i. e., in the signal handler) and that the data and memory consistency guarantees hold. The variables manipulated within the signal handler (i. e. those that must be declared global) are integers – often just loop variables – and typically not numerous. If they appear often (i. e. also elsewhere in the code) and a user would like them to remain local there, he can rename the variables and make them global for

those segments of the code whose “execution” must be interruptible when a user changes something.

In light of the interface of the application to the framework, only negligible effort is needed to register new variables for steering. However, to complete the picture, the possible overall effort from a user’s point of view to exploit the idea of computational steering has to be discussed. Therefore, what follows is a brief overview, based on the experience with the scenarios explored thus far.

A distinction is made, first of all, among non-hierarchical (direct) and hierarchical (adaptive) approaches. Due to the lack of generality for the variety of simulation scenarios, the latter approaches are not assumed to be a responsibility of the framework. However, they are investigated and exploited here for each specific simulation. The framework has proved itself to be flexible enough to support at least several different hierarchical simulation concepts.

Most straightforward is steering of the parameters such as the maximal number of iterations within the computation, accuracy requirements for an iterative solver, etc.

The framework also directly supports steering of any variables where a user does not need to reuse values from the previous calculation – in other words, when the next iteration in the interactive computing loop starts from scratch. The data, however, would have to be brought into the initial state before the interactive computing loop. Nevertheless, not all steps, like memory allocation, would have to be repeated due to the user update, especially in the case of time- and memory-consuming pre-processing phases in comparison to the rest of the computation. Rather, only the necessary reinitialisation would take place within the interactive computing loop.

Another steering category is where the required reinitialisation routines involve some discretisation parameters, which are used in numerical modelling but, nevertheless, do not affect the underlying primary mesh or the dimensions of the re-used data, including the intermediate solution.

Finally, the most challenging scenario is steering the parameters that affect, for instance, the solution-related data and its dimensions, such as a mesh resolution. Like in the upsampling procedure, it might be required – as an initial guess – to assign a smaller number of entries of the calculated data on the coarser geometry mesh to the larger number of entries.

In a distributed environment, the non-blocking communication routines provided by the framework have to be employed. However, the framework cannot determine how the data sent to the front-end should be matched to the visualisation for all the diverse application scenarios.

Chapter 5

Summary

It is essential to be able to efficiently analyse and react in real-time to large data sets from highly accurate simulations of phenomena. The immersive experience is possible today due to advances in software methods and hardware technologies, leading to intuitive exploration of the desired phenomena, the ability to prove or negate postulates, and also to learning by chance. Not only experts with good comprehension of numerical simulation methods, efficient algorithms, high-performance computing, networking, or visualisation, but also experts in all the other fields – medical doctors, or architects, for example – need to take a central role in analysis to make more informed decisions.

For this, they need to be equipped with tools – consisting of a simulation back-end, and a visualisation/UI front-end – whose complexity is hidden from the end-user. Both front- and back-end components should be computationally efficient. The major challenge when implementing these tools is to connect front- and back-end components and make a running simulation instantly aware of interaction on the user side concerning, for example, boundary conditions or any parameter tuning, so that it automatically starts a new computation with the actual settings.

State-of-the-art attempts to achieve this have resulted either in the insertion of check- and break-points at fixed places in the simulation code or in dedicating one thread per process only to checking for updates. What is needed is a generic, minimally invasive concept, which would make interactive computing widely applicable. However, the conventional methods can, depending on the size of the problem, still compromise the benefits of the asynchronous computations and the exchange between simulation and visualisation; and, in the latter case, they can lead to logistical issues.

In this study, a generic framework is presented based on a "minimal invasion" principle (i. e. minor code changes necessary) and coupling simulation codes and visualisation tools. It allows a user to trigger a simulation during the run time and receive prompt feedback. However, the use of signals to provide a high level of interactivity in diverse application scenarios and for different problem sizes has

also introduced many challenges to keep memory and data consistency, which result from asynchronous interrupts.

Four application scenarios are illustrated and discussed in terms of the overhead of the framework, it's responsiveness to changes, and also the necessary effort from users or developers to integrate it. Some of these examples also show how improving the application basis – using hierarchical approaches, optimal parallelisation techniques, etc. – can make the overall interactive process faster and more intuitive. The outcome conclusions, possibilities for extension of the features and the perspective for future work in general are all summarised in this final chapter.

5.1 Conclusions

Coupling components together into an efficient, intuitive, and interactive computing environment demands first thorough consideration of each component separately, as proved in all the application scenarios described in this work. Without efficient simulation, the interactive computing process is slowed down tremendously. Especially when it comes to large scale simulations, approximation methods and fast solvers based on hierarchies, where applicable, are rather a prerequisite to achieve a fast interactive trial-and-error process, particularly in combination with an adequate parallelisation strategy. What is required for an effective parallel computation approach are the following hardware and software properties: rapid communication between the individual processes, fast data transfer to and from memory, a protocol based on ports or on process identifiers, methods to reduce the demand on the network between all the application components and effective decomposition of the problem, allowing for the optimal load balancing for the system. Real-time scientific visualisation, although not itself part of this work, plays a major role in the intuitive interpretation of results. Thus, in the case of large amounts of data, selective updates of only the part of the domain of interest – with adopted resolution, etc. – might be desirable. Moore's law in software technology is the main prerequisite for exploiting forthcoming powerful hardware. Moreover, it is often necessary to rethink and rewrite the existing scientific and engineering community codes, to make them more efficient, extensible, and reusable. This, however, is a large step. Especially if the user or developer did not write the initial application code, the integration of the framework is faster and "smoother" for well-structured and easily extensible codes. This applies to all four described application cases.

Since the concept of signals is used in this work, this concept is discussed in light of different codes (sequential or parallel) or different operating systems (Unix or Windows) and in cases where different compilers are required. In addition to the combinations of signals with multithreading, which involve special treatment, challenges also arise from ensuring memory and data consistency during sudden interrupts. Asynchronous but reliable communication patterns between the user front-end and simulation back-end and fast data transfer, based on the selection of the desired part of the domain, need to be tackled as well. Finally, hierarchical simulation approaches are desirable in general and are more or less effective in

conjunction with the framework. These approaches are neither technically a part of nor a responsibility of the framework; however, they are supported and tested in most of the application test cases. Based on the promising results, it is advisable to use these approaches whenever possible.

It would be pretentious to claim that the concept is best suited for every application; however, it is well suited for many of them today. It provides an easy-to-integrate functionality with a minimally invasive interface for a user, whose field of expertise, or focus, may differ from real-time interactive computing. Its applicability to a wide range of problems and problem sizes – without major implementation-related distinctions among them – makes it different from other state-of-the-art tools. This point is illustrated by the four application cases in which the framework was integrated. Those are the codes from different scientific and engineering communities and institutes, which are diverse in regards to simulation and visualisation methods used, design patterns, programming languages, compilers used, etc. The results in terms of the overhead of the framework, and the user effort to integrate it, are presented for all the test applications and also for different problem sizes. Where possible, hierarchical and parallelisation approaches were applied to test and propose optimal application bases. The possible overall effort from a user's point of view to exploit the interactive computing framework is determined by the complexity of necessary reinitialisation routines. Apart from the reinitialisation, if a simulation runs sequentially or is parallelised using shared-memory concepts, the rest of the effort is negligible. In a distributed environment, the non-blocking communication provided by the framework has to be employed in addition. Determining how the data sent to the front-end should be matched to the visualisation, however, cannot be assumed by the framework due to the diversity of application scenarios.

5.2 Outlook

Although the results for the first application cases look very promising, further research should be done on the question of the signal transfer from a user to all the computing nodes in the case of massively parallel simulation. Furthermore, the framework should be integrated and tested in more – especially distributed – parallel engineering simulation scenarios. Hierarchical approaches can be tested for h-version FEM and a variation of the current multi-level version for FD – namely, those that run several instances of the simulation program for different h values in both cases, and also with the framework integrated. Existing features of the sliding window approach for fast data transfer should be extended. Regarding the implementation for Windows OS, the simulation has performed well in initial small, rather artificial test cases. To further test its suitability in this context, it would be useful to incorporate it into more "real life" applications that presently exist in engineering communities. The very "nature" of the work described, which has fortunately involved much collaboration and diverse interdisciplinary projects, has led to fruitful discussions about how the framework can evolve some day into a "next generation" interactive computing tool. Needs of

different scientific and engineering communities and projects led to a large pool of ideas for extension of the supported features. The challenge will certainly remain how to put them all together – constantly adding new ones – while keeping the generality and without significantly compromising the efficiency.

Appendix: Code Modifications

This appendix provides segments of the code from the 2D heat conduction simulation described in Chapter 4. The intention is to present the code changes which were necessary in order to integrate the proposed concept of the framework. These modifications (*MOD. #*) are marked correspondingly. The code contains also modifications related to the hierarchical approach for this test scenario, which are technically not a responsibility of the framework itself. Parts of the code less relevant for this "illustration" are either completely excluded or simplified for the sake of clarity (such as those related to visualisation and user interaction).

```
/*************************************************
 *                 INCLUDED HEADERS              *
 *************************************************/
#inlcude <queue>
#include <csignal>
#include <mpi.h>

/*************************************************
*             USER HEADERS AND DEFINITIONS       *
*************************************************/
#include <MyEvent.h> // this is where a data structure
// for user event data is stored
#define BUFFER 10000 // size of the buffer for Bsend

using namespace std;

// MOD. 1: Variables related to signal (declared global)
volatile sig_atomic_t itermax;    // limit for a loop index iter.
volatile sig_atomic_t iter1 = 0; // loop index to be manipulated

// Variables used for hierarchical approach
volatile sig_atomic_t activity = 0; // user activity "index"
volatile bool interaction = false; // indicator
volatile long seconds; // to calc. freq. of interaction
volatile long seconds\_old;
volatile sig_atomic_t level; // "hierarchy" id
```

```
// Variables related to user interaction
queue< MyEvent* > qx; // queue where user events are
// stored to prevent thread hazards
long change[3]; // buffer where user data is received

// MOD. 2: Global variables related to MPI
MPI_Status status;
double res[BUFFER]; // result buffer

/**************************************************
*                SIMULATION MAIN FUNCTION          *
**************************************************/
// initialisation steps
interactive_membraneFrm* frame;
frame = new interactive_membraneFrm(NULL);
activity = 2; // an indicator of the highest hierarchy
level = 0;    // the finest grid

// MOD. 3: Initialisation of sigaction and setting alarm
struct sigaction act, oact;
act.sa_handler = alarmHandler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
#ifdef SA_RESTART
  act.sa_flags |= SA_RESTART;
#endif
sigaction(SIGALRM, &act, &oact);
ualarm(1000, 0);

// the interactive computing loop
while(1) {frame->computeMembrane();}

/* End of the main function */

/**************************************************
* IMPLEMENTATION OF THE SIGALRM HANDLER *
**************************************************/
void alarmHandler (int p) // MOD. 4
{
 int flag = 1; // indicator of a received message
 long x, y; // coordinates of the changed entity
 char t;    // type of the changed entity

 if(pthread\_mutex\_trylock( &mutex2 )){
   ualarm(1000, 0);
```

```
} // synch. with the user thread
else { // lock obtained
  interaction = 0;
  while(flag) { // Receive all the user events\\
    MPI_Iprobe (0, MPI_ANY_TAG, MPI_COMM_WORLD,
                &flag, &status);
    // the time when the event was sent is provided via:
    seconds = status.MPI\_TAG;
    if(flag){// if there was a message...\\
      MPI_Recv (&change, 3, MPI_LONG, 0, seconds,
                MPI_COMM_WORLD, &status);
      // store a new event in the queueldots
          t = change[0]; x = change[1];  y = change[2];
          qx.push(new MyEvent1(x, y, t));
          interaction = 1;
    }
    else {}
  } // while(flag)

  // NEEDED ONLY FOR HIERARCHICAL APPROACH:
  if (interaction) {
    iter1 = itermax;
    interaction = false;

    // indicator of the frequency of users activity
    if (seconds < 10000) { activity = 0; }
      else { activity = 1; }
      seconds_old = seconds;
  }
  else { // there was no user interaction
    struct timeval now;
    gettimeofday(&now, NULL);

    if( ((now.tv_sec * 1000000 + now.tv_usec)
          - seconds_old) > 100000)
      { activity = 2; }
    else {
      iter1 = itermax;
      activity = 1;
        }
  }// no user interaction

   ualarm(1000, 0);
 } // finished
}
/* END OF SIGALRM HANDLER */
```

```
/***********************************************
*  void interactive_membraneFrm::computeMembrane() *
***********************************************/
{
  while (qx.size()>0) {
   if(qx.front()->type() == 'a') { // new heat source
      int n = this->WxGridPillar->GetRows();
      this->WxGridPillar->AppendRows(1);

      wxString sx,sy,sz;
      sx << qx.front()->x();
      sy << qx.front()->y();
      sz << 50;
      this->WxGridPillar->SetCellValue(n,0,sx);
      this->WxGridPillar->SetCellValue(n,1,sy);
      this->WxGridPillar->SetCellValue(n,2,sz);
      qx.pop(); // remove this event from the queue
    }
    else { ... } // other types of interaction
  }  // user thread can obtain the 'lock' meanwhile
  /* Hierarchy is determined */
  memb->nx = this->getNXGrid();
  memb->ny = this->getNYGrid();
  memb->rasterizeBoundaries();

  /* A call to Gauss-Seidel solver */
  memb->iterateDisplacement();
} // End of computeMembrane()

/***********************************************
*                 GAUSS-SEIDEL SOLVER            *
***********************************************/
int Membrane::iterateDisplacement()
{
 int i, ix, jx, x1, y1; // loop indexes
 double u_old;
 double local_error, error;
 double eps = wx->getTolerance();
 itermax = wx->getMaxIterations();
 // get grid spacings
 double hx = 300.0/(double)(nx);
 double hy = 300.0/(double)(ny);
 // set the pillars
 for (i=0;i<wx->getSizePillar();i++) {
   x1 = (int)( wx->getXPillar(i)/hx + 0.5);
```

```
    y1 = (int)( wx->getYPillar(i)/hy + 0.5);
    if (x1<0 || x1>nx-1) continue;
    if (y1<0 || y1>ny-1) continue;

    // Determine the needed "hierarchy"
    if(nx ==_nx && ny ==_ny) {
          grid[x1][y1] = 1;
          u[x1][y1] = wx->getZPillar(i);
    }
    else { // NEEDED ONLY FOR HIERARCHICAL APPROACH
         grid1[level][x1][y1] = 1;
         u1[level][x1][y1] = wx->getZPillar(i);
    }
  } // end of for

  for (iter1=0; iter1<itermax; iter1++) {
    error = 0.0;
    // loop over all grid points
    for (ix=1; ix<nx-1; ix++) {
      for (jx=1; jx<ny-1; jx++) {
        // compute new displacement, compare to the old one
        if (nx ==_nx) { // the finest "hierarchy"
          // (check here if the point is inside the domain)
          u_old = u[ix][jx];
          u[ix][jx] = (u[ix][jx+1] + u[ix+1][jx]
                        + u[ix][jx-1] + u[ix-1][jx])/4.0;
          local_error = fabs(u[ix][jx]-u_old);
          if (local_error > error) error = local_error;
        }
        else { // NEEDED ONLY FOR HIERARCHICAL APPROACH
          // (check here if the point is inside the domain)
          u_old = u1[level][ix][jx];
          u1[level][ix][jx] = (u1[level][ix][jx+1]
                               + u1[level][ix+1][jx]
                               + u1[level][ix][jx-1]
                               + u1[level][ix-1][jx])/4.0;
          local_error = fabs(u1[level][ix][jx]-u_old);
          if (local_error > error) error = local_error;
        }
      }
    }
    // pack result into comm. buffer (user's task)
    int s = 0;
    for (int s1=0; s1<=nx-1; s1++) {
      for (int s2=0; s2<=ny-1; s2++) {
        if (nx == _nx)
```

```
          res[s++] = u[s1][s2];
        else // NEEDED ONLY FOR HIERARCHICAL APPROACH
          res[s++] = u1[level][s1][s2];
      }
    } // MOD. 5: ... and send the result to the user
    MPI_Ssend (&res[0], nx*ny, MPI_DOUBLE, 0, 2,
                        MPI_COMM_WORLD);
    // check if the tolerance is met
    if (error < eps) { break; }
  }
}
/* End of Gauss-Seidel solver */

/*********************************************
 *          USER INTERACTION PROCESS          *
 *********************************************/
 queue< MyEvent* > qx; // queue where user events are
 long change[3];       // send buffer

// Iterative sending of the user events' data
while (qx.size() > 0) {
   // store an event in the send buffer
  if(qx.front()->type() == 'a') { // new heat source
    change[0]= 'a';
    change[1] = qx.front()->x();
    change[2] = qx.front()->y();
    qx.pop(); // remove an event from the queue
  }
  else { ... } // process any other event

  // MOD. 6: send an update to the simulation
  MPI_Bsend(&change[0], 3, MPI_LONG, 1, seconds,
            MPI_COMM_WORLD);
}
```

Bibliography

[1] SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI).

[2] Wikipedia – Ischaemic heart disease, url: http://en.wikipedia.org/wiki/Ischaemic_heart_disease.

[3] Wikipedia – Defibrillation, url: http://en.wikipedia.org/wiki/Defibrillation.

[4] Institut für Computeranwendungen im Bauingenieurwesen der Technischen Universität Braunschweig, url: http ://www.bau-ings.de/flowsim/.

[5] Enzo, url: http://lca.ucsd.edu/portal/software/enzo.

[6] The Center for the Simulation of Accidental Fires and Explosions, url: http://www.csafe.utah.edu.

[7] Argon Design, url: http://www.argondesign.com/news/2012/sep/11/multicore-many-core, 2012.

[8] TCP Guide, url: http://www.tcpipguide.com/free/index.htm, 2012.

[9] 3dfx Interactive Archive, url: http://3dfx.com/ (arch.).

[10] NVIDIA: What is GPU Computing, url: http://www.nvidia.com/object/what-is-gpu-computing.html, 2012.

[11] NVIDIA Kepler, url: http://www.nvidia.com/object/ nvidia-kepler.html.

[12] NVIDIA: Dynamic Parallelism, url: http://developer.download.nvidia.com, 2012.

[13] NVIDIA Applications Catalogue, url: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf, 2012.

[14] COVISE project, url: http://www.hlrs.de/organization/av/vis/covise/.

[15] Orbacus CORBA library, url: http://orbacus.com.

[16] Vampir project, url: http://www2.fz-juelich.de/jsc/cv/tools/vampir.

[17] Qt Project. http://qt-project.org.

[18] wxWidgets Library. www.wxwidgets.org.

[19] *Finite Element Analysis*. John Wiley and Sons Inc., New York, 1991.

[20] Reality Grid, 2003. Reality Grid project: www.realitygrid.org.

[21] 2004. CUMULVS software, url: www.csm.ornl.gov/cs/cumulvs.html.

[22] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 2004.

[23] 2011. Report, url: https://www.xsede.org/home/-/journal_content /56/18157/158815?refererPlid=18161.

[24] 2012. Wikipedia – Steered Molecular Dynamics, url: http://en.wikipedia.org/wiki/Molecular_dynamics.

[25] 2012. Wikipedia – Cryosection, url: http://en.wikipedia.org/wiki/ Frozen_section_procedure.

[26] 2012. Wikipedia – Design Pattern, url: http://en.wikipedia.org/wiki/ Software_design_pattern.

[27] 2012. Linux Manual, url: http://www.kernel.org/doc/man-pages.

[28] 2012. GNU C Manual, url: http://www.gnu.org/software/libc/manual.

[29] Parallel Environment Runtime Edition: MPI Programming Guide SC23-6783-04, 2012.

[30] TurboPower Sleuth QA Suite, 2012. url: http://turbopower-sleuth-qa-suite.software.informer.com.

[31] Visible Human Project, 2012. url: http://www.nlm.nih.gov/research/visible/ visible_human.html.

[32] AMD Radeon HD 7970 Graphics, 2013. http://www.amd.com.

[33] J. P. Ahrens, J. Woodring, D. E. DeMarle, J. Patchett, and M. Maltrud. Interactive Remote Large-Scale Data Visualization via Prioritized Multi-Resolution Streaming. In *Proceedings of the 2009 Workshop on Ultrascale Visualization*, UltraVis '09, pages 1–10, New York, NY, USA, 2009. ACM.

[34] G. Allen, T. Goodale, and E. Seidel. The Cactus Computational Collaboratory: Enabling Technologies for Relativistic Astrophysics, and a Toolkit for Solving PDE's by Communities in Science and Engineering. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 36–, Washington, DC, USA, 1999. IEEE Computer Society.

[35] M. Ament, S. Frey, F. Sadlo, T. Ertl, and D. Weiskopf. GPU-based 2D Flow Simulation Steering using Coherent Structures. In *Proceedings of the 2nd International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, volume 2011, page paper18, 2011.

[36] A. Atanasov, H.-J. Bungartz, J. Frisch, M. Mehl, R.-P. Mundani, E. Rank, and C. van Treeck. Computational Steering of Complex Flow Simulations. In *High Performance Computing in Science and Engineering in Garching/Munich 2009*, pages 63–74, 2010.

[37] A. Atanasov and T. Weinzierl. Query-Driven Multiscale Data Processing in Computational Fluid Dynamics. In *Proceedings of the International Conference on Computational Science, ICCS 2011*, volume 4, pages 332–341, 2011.

[38] S. Auer, C. B. Macdonald, M. Treib, J. Schneider, and R. Westermann. Real-Time Fluid Effects on Surfaces using the Closest Point Method. *Computer Graphics Forum*, 2012. To appear.

[39] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Sliding Window Computations over Data Streams. Technical Report 2002–25, Stanford Infolab, 2002.

[40] M. Banahan. *The C Book*. Addison Wesley, 2003.

[41] U. Baneerje. *Dependence Analysis*. Kluwer Academic Publishers, 1997.

[42] E. Bänsch and W. Dörfler. Adaptive Finite Elements for Exterior Domain Problems. *Numer. Math.*, 80:497–523, 1998.

[43] G. Baolai. Fortran Signal Handling. In *Proceedings of the SHARCNET Workshop*, 2008.

[44] R. G. Belleman and Sloot P. M. A. The Design of Dynamic Exploration Environments for Computational Steering Simulations. In *Proceedings of the SGI Users' Conference*, pages 57–74, 2000.

[45] G. Biermann and F.-J. Kalze. Helios – Computer Aided Lighting, The Path from Simulation to Prototype. In *Proceedings of 29th ISATA Conference*, Florenz, 1996.

[46] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe. Multilevel Streaming for Out-of-Core Surface Reconstruction. In *Proc. of the 5th Eurographics Symposium on Geometry Processing (SGP'07), Eurographics Association*, pages 69–78, 2007.

[47] A. Borrmann, P. Wenisch, M. Egger, C. van Treeck, and E. Rank. Collaborative Computational Steering: Interactive Collaborative Design of Ventilation and Illumination of Operating Theatres. In *Proc. Intelligent Computing in Engineering (ICE 2008)*, 2008.

[48] A. Borrmann, P. Wenisch, and C. van Treeck. Collaborative HVAC Design Using Interactive Fluid Simulations: A Geometry-Focused Platform. In *Proceedings of 12th International Conference of Concurrent Engineering*, Fort Worth, TX, USA, 2005.

[49] A. Borrmann, P. Wenisch, C. van Treeck, and E. Rank. Collaborative Computational Steering: Principles and Application in HVAC Layout. *Integrated Computer-Aided Engineering*, 13:1–16, 2006.

[50] A. Brandt. *Multigrid Guide with Applications to Fluid Dynamics.* GMD-Studie 85, 1985.

[51] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial, 2nd ed.* SIAM Press, 2000.

[52] K. Brodlie, J. Wood, D. Duce, and M. Sagar. gViz: Visualization and Computational Steering on the Grid. In *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 54–60, 2004.

[53] J. E. Brooke, P. V. Coveny, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational Steering in RealityGrid. In *Proceedings of UK e-Science All Hands Meeting*, 2003.

[54] S. Bryson and C. Levit. The Virtual Wind Tunnel. *IEEE Computer Graphics and Applications*, 12:25–34, 1992.

[55] H.-J. Bungartz and M. Griebel. Sparse Grids. *Acta Numerica*, 13:147–269, 2004.

[56] H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. The PDE Framework Peano Applied to Fluid Dynamics: An Efficient Implementation of a Parallel Multiscale Fluid Dynamics Solver on Octreelike Adaptive Cartesian Grids. *Computational Mechanics*, 46:103–114, 2010.

[57] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[58] B. M. Burton, J. D. Tate, B. Erem, D. J. Swenson, D. F. Wang, M. Steffen, D. H. Brooks, P. M. Van Dam, and R. S. Macleod. A Toolkit for Forward/Inverse Problems in Electrocardiography within the SCIRun Problem Solving Environment. In *Proceedings of Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 267–270, 2011.

[59] D. Butnaru, D. Pflüger, and H.-J. Bungartz. Towards High-Dimensional Computational Steering of Precomputed Simulation Data using Sparse Grids. *Procedia Computer Science*, 4(0):56–65, 2011.

[60] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[61] J. Davison de St. Germain, J. J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah : a Massivelly Parallel Problem Solving Environment. In *Proceedings of the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41, 2000.

[62] T. De Fanti. Visualization in Scientific Computing. *Computer Graphics*, 21(6), 1987.

[63] M. H. de Vaal, J. Neville, M. Litow, J. Scherman, P. Zila, and T. Franz. Patient-Specific Prediction of Intrinsic Mechanical Loadings on Sub-Molecular Pectoral Pacemaker Implants Based on an Inter-Species Transfer Function. *Journal of Biomechanics*, 44(14):2525–2531, 2011.

[64] R. H. Desmond, L. Renambot, H.E. Bal, D. Germans, and H. J. W. Spoelder. CAVEStudy: an Infrastructure for Computational Steering in Virtual Reality Environments. In *Proceedings of the Ninth IEEE International Simposium on High Performance Distributed Computing*, pages 57–61. IEEE Computer Society Press, 2000.

[65] C. Dick. *Computational Steering for Implant Planning in Orthopedics*. Dissertation, Technische Universität München, München, 2012.

[66] C. Dick, R. Georgii, R. Burgkart, and R. Westermann. Computational Steering for Patient-Specific Implant Planning in Orthopedics. In *Proceedings of Visual Computing for Biomedicine*, pages 83–92, 2008.

[67] C. Dick, R. Georgii, R. Burgkart, and R. Westermann. Stress Tensor Field Visualisation for Implant Planning in Orthopedics. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1399–1406, 2009.

[68] W. Dörfler. A Convergent Adaptive Algorithm for Poisson's Equation. *SIAM J. Numer. Anal.*, 33:1106–1124, 1996.

[69] F. W. Dorr. The Direct Solution of the Discrete Poisson Equation on a Rectangle. *SIAM Rev.*, 12(2), 1970.

[70] A. Düster and J. Frisch. Environment for interactive exploration of 2d heat conduction, 2008. Advanced Computational Methods 2, Master program Computational Mechanics at TUM (www.come.tum.de).

[71] A. Düster, A. Niggl, and E. Rank. Applying the hp-d Version of the FEM to Locally Enhance Dimensionally Reduced Models. *Computer Methods in Applied Mechanics and Engineering*, pages 3524–3533, 2007.

[72] A. Düster, Z. Parvizian, Z. Yang, and E. Rank. The Finite Cell Method for Three-Dimensional Problems of Solid Mechanics. In *Proceedings of Computer Methods in Applied Mechanics and Engineering*, pages 3768–3782, 2009.

[73] A. Düster and E. Rank. A p-Version Finite Element Approach for Two- and Three-Dimensional Problems of the J2 Flow Theory with Nonlinear Isotropic Hardening. *International Journal for Numerical Methods in Engineering*, 53:49–63, 2001.

[74] T. Fogal and Krüger J. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modelling, and Visualization*, 2010.

[75] T. Forkert, H.-P. Kersken, A. Schreiber, M. Strietzel, and K. Wolf. The Distributed Engineering Framework TENT. In *VECPAR*, pages 38–46, 2000.

[76] J. Frisch, V. Varduhn, J. Knezevic, M. Pfaffinger, M. Egger, R. Mundani, and E. Rank. Interactive Thermal Comfort Assesment Simulations. In *High Performance Computing in Science and Engineering in Garching/Munich 2012*, pages 96–98, 2012.

[77] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, 1997.

[78] J. A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. on Numer. Analysis*, 10:345–363, 1973.

[79] R. Goering. Panel Confronts Multicore Pros and Cons.

[80] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : Portable Parallel Programming With the Message-Passing Interface Scientific and Engineering Computation*. MIT Press, 1999.

[81] The Open Group. The Single UNIX Specification, Version 2, 2007.

[82] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer, 1985.

[83] C. D. Hansen and C. R. Johnson. *The Visualization Handbook*. Elsevier, 2005.

[84] Ch. Hirsch. *Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics, Second Edition*. Elsavier, 2007.

[85] T. Holwerda. Intel: Software Needs to Heed Moore's Law, 2007.

[86] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Publications, 2000.

[87] H. Izakian, A. Abraham, and V. Snášel. Comparison of Heuristics for Scheduling Independent Tasks on Heterogeneous Distributed Environments. In *Proc. of International Joint Conference on Computational Sciences and Optimization*, volume 1, pages 8–12, 2009.

[88] D. J. Jablonowski. VASE: The visualisation and application steering environment. In *Proc. of Supercomputing '93.*, pages 560–569, 1993.

[89] D. Jenz and M. Bernreuther. The Computational Steering Framework Steereo. In *Proceedings of PARA 2010 Conference: State of the Art in Scientific and Parallel Computing*, 2010.

[90] T. Jevremovic, T. Itoa, and Y. Inabaa. ANEMONA: Multiassembly Neutron Transport Modeling. *Annals of Nuclear Energy*, 29:2105–2125, 2002.

[91] T. Jevremovic, H. C. Lee, K. Retzke, Y. Peng, and M. Hursin. AGENT Code: Open-Architecture Analysis and Configuration of Research Reactors—Neutron Transport Modeling with Numerical Examples. In *Proceedings of PHYSOR—The Physics of Fuel Cycles and Advanced Nuclear Systems: Global Developments*, Chicago, Illinois, 2004.

[92] C. R. Johnson. Biomedical Visual Computing: Case Studies and Challenges. *IEEE Computing in Science and Engineering*, 14(1):12–21, 2012.

[93] C. R. Johnson and S. G. Parker. A computational steering model applied to problems in medicine. *Supercomputing 94*, pages 540–549, 1994.

[94] C. R. Johnson, S G Parker, C. D. Hansen, G. L Kindlmann, and Y. Livnat. Interactive simulation and visualization. *IEEE Computer*, 32(12):59–65, 1999.

[95] C. R. Johnson, S. G. Parker, D. Weinstein, and S. Heffernan. Component-based problem solving environments for large-scale scientific computing. *Conc. Comp.: Prac. Exper.*, 14:1337–1349, 2002.

[96] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[97] D. S. Katz1, S. Callaghan, R. Harkness, S. Jha, K. Kurowski, S. Manos, S. Pamidighantam, M. Pierce, B. Plale, C. Song, and J. Towns. Science on the TeraGrid. *COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY*, Special Issue 2010:81–97, 2010.

[98] T. Kesavadas and A. Sudhir. Computational Steering in Simulation of Manufacturing Systems. In *ICRA*, pages 2654–2658, 2000.

[99] K. D. Kima and Rizwan-uddin. A Web-Based Nuclear Simulator using RELAP5 and LabVIEW. *Nuclear Engineering and Design*, 237:1185–1194, 2007.

[100] J. Knežević, R-P. Mundani, E. Rank, A. Khan, and C. R. Johnson. Extending the SCIRun Problem Solving Environment to Large-Scale Applications. In *Proc. of The IADIS Applied Computing 2012*, 2012.

[101] J. Knezevic. Dynamic Load Balancing in Structure Simulation, 2009.

[102] J. Knezevic, J. Frisch, R.-P. Mundani, and E. Rank. Interactive Computing Framework for Engineering Applications. In *Proceedings of the INTERCOMP 2011—International Conference on Computer Science and Applied Computing*, 2011.

[103] J. Knezevic, J. Frisch, R.-P. Mundani, and E. Rank. Interactive Computing Framework for Engineering Applications. *Journal of Computer Science*, 7(5):591–599, 2011.

[104] J. Knezevic, H. Hernandez, T. Fogal, and T. Jevremovic. Visual Simulation Steering for a 3D Neutron Transport AGENT Code System. In *Proceedings of INREC International Nuclear and Renewable Energy Conference*, 2012.

[105] J. Knezevic and R.-P. Mundani. Applying Dynamic Load Balancing Techniques to the Parallel p-Version FEM Using Nested Dissection. In *Proceedings of 21st Forum Bauinformatik*, pages 105–113, 2009.

[106] J. Knezevic and R.-P. Mundani. Interactive Computing for Engineering Applications. In *Proceedings of the 22nd Forum Bauinformatik*, pages 137–144, 2010.

[107] J. Knezevic, R.-P. Mundani, and E. Rank. Interactive Computing—Virtual Planning of Hip Joint Surgeries with Real-Time Structure Simulations. *International Journal of Modeling and Optimization*, 1(4):308–313, 2011.

[108] J. Knezevic, R.-P. Mundani, and E. Rank. Interactive Computing in Preoperative Planning of Joint Replacement. In *Proceedings of the International Conference on Modeling, Simulation and Control*, volume 10, pages 86–91, 2011.

[109] J. Knezevic, R.-P. Mundani, and E. Rank. Schedule Optimisation for Interactive Parallel Structure Simulations, 2012.

[110] J. Knezevic, R.-P. Mundani, E. Rank, H. Hernandez, T. Jevremovic, and T. Fogal. Interactive Computing in Numerical Modelling of Particle Transfer Methods. In *Proceedings of IADIS International Conference on Theory and Practice in Modern Computing*, 2012.

[111] H. Kobashi, S. Kawate, Y. Manabe, M. Matsumoto, H. Usami, and D. Barada. PSE Park: Framework for Problem Solving Environments. *Journal of Convergence Information Techonolgy*, 5(4):225–239, 2010.

[112] J. Kopf, M. F. Cohen, and M. Lischinski, D. ad Uyttendaele. Joint Bilateral Upsampling. *ACM Trans. Graph*, 26(3), 2007.

[113] E. Kraemer and J. Vetter. Computational Steering. *Hawaii International Conference on System Sciences*, 7:126, 1998.

[114] D. Krishnan and R. Szeliski. Multigrid and Multilevel Preconditioners for Computational Photography. In *Proceedings of SIGGRAPH Asia 2011*, Hong Kong, 2011.

[115] S. Kühner, P. Hardt, M. Krafczyk, and E. Rank. Computational Steering of a Lattice-Boltzmann based CFD-Solver in Virtual Reality. *CONVR 2003*, 2003.

[116] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4), 1999.

[117] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[118] J. Leech, J. F. Prins, and J. Hermans. SMD: Visual Steering of Molecular Dynamics for Protein Design. *IEEE Computational Science and Engineering*, 3(4):38–45, 1996.

[119] D. Levine, M. Facello, P. Hallstrom, G. Reeder, B. Walenz, and F. Stevens. Stalk: An Interactive System for Virtual Molecular Docking. *IEEE Computational Science*, 4(2):55–65, 1997.

[120] V. Mann, R. Matossian, R. Muralidhar, and M. Parashar. DISCOVER: An Environment for Web-Based Interaction and Steering of High-Performance Scientific Applications. *Concurrency—Practice and Experience*, 13:737–754, 2001.

[121] R. Marshall, J. Kempf, S. Dyer, and C.-C. Yen. Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie. *SIGGRAPH Comput. Graph.*, 24(2):89–97, 1990.

[122] P.-G. Martinsson. *Fast Multiscale Methods for Lattice Equations*. PhD thesis, The University of Texas at Austin, 2002.

[123] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in Scientific Computing. *Computer Graphics*, 21(6), 1987.

[124] M. Miller, C. D. Hansen, S. G. Parker, and C. R. Johnson. Simulation steering with scirun in a distributed memory environment. In *Proceedings of Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, 1998.

[125] M. Miller, C. Moulding, J. Dongarra, and C. R. Johnson. Grid-enabling problem solving environments: A case study of scirun and netsolv. In *Proceedings of The 5th International Conference on High-Performance Computing, 2001 Advanced Simulation Technologies Conference, Society for Modeling and Simulation International*, pages 98–103, 2001.

[126] J. D. Mulder, R. van Liere, and J. J. van Wijk. Computational Steering in the CAVE. *Future Generation Comp. Syst.*, 14(3-4):199–207, 1998.

[127] J. D. Mulder, J. J. van Wijk, and R. van Liere. A Survey of Computational Steering Environments. *Future Gener. Comput. Syst.*, 15(1):119–129, feb 1999.

[128] J.D. Mulder and J.J. Van Wijk. Logging in a Computational Steering Environment. In *Visualization in Scientific Computing '95, Proceedings of the sixth Eurographics Workshop*, pages 118–125. Springer Verlag, 1995.

[129] R.-P. Mundani, H.J. Bungartz, E. Rank, A. Niggl, and R. Romberg. Extending the p-Version of Finite Elements by an Octree-Based Hierarchy. In O. Widlund and D.E. Keyes, editors, *Domain Decomposition Methods in Science and Engineering XVI*, volume 55 of *Lecture Notes in Computational Science and Engineering*, pages 699–706. Springer, 2007.

[130] R.-P. Mundani, A. Düster, J. Knezevic, A. Niggl, and E. Rank. Dynamic Load Balancing Strategies for Hierarchical p-FEM Solvers. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes on Computer Science*. Springer, 2009.

[131] A. Muthler, D. Scholz, A. Düster, W. Volk, M. Wagner, and E. Rank. Adaptive p-FEM for Spring Back Analysis. In *Proc. of the 5th International Conference on Computation of Shell and Spatial Structures*, Salzburg, Austria, 2005.

[132] R. Nicolas, A. Esnard, and O. Coulaud. Toward a Computational Steering Environment for Legacy Coupled Simulations. In *Proceedings of Sixth International Symposium on Parallel and Distributed Computing*, 2007.

[133] V. Nübel, A. Düster, and E. Rank. An rp-Adaptive Finite Element Method for Elastoplastic Problems. In *Proc. of European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS)*, 2004.

[134] S. Parker, M. Miller, C. D. Hansen, and C. R. Johnson. An Integrated Problem Solving Environment: The SCIRun Computational Steering System. In *In Hawaii International Conference of System Sciences*, pages 147–156. IEEE Computer Society, 1998.

[135] S. G. Parker, D. M. Beazley, and C. R. Johnson. Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59, 1997.

[136] S.G. Parker. The SCIRun Problem Solving Environment and Computational Steering Software System, 1999. Parker PhD thesis, University of Utah, url: http://www.cs.utah.edu/~sparker/publications/thesis.pdf.

[137] V. Pascucci and Frank R. J. Hierarchical Indexing for Out-of-Core Access to Multi-Resolution Data. *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 225–241, 2002.

[138] M. Pfaffinger, C. van Treeck, A. Borrmann, P. Wenisch, and E. Rank. An Interactive Thermal Fluid Simulator for the Design of HVAC systems. In *Proceedings of ASCE Int. Workshop in Computing in Civil Engineering*, Pittsburgh, USA, 2007.

[139] R. Piché. Partial Differential Equations. url: http://math.tut.fi/~piche/pde/notes01.pdf.

[140] S. M. Pickles, R. Haines, R. L. Pinning, , and A. R. Porter. Computational Steering in RealityGrid. In *Proceedings of UK e-Science All Hands Meeting*, 2004.

[141] B. Plale, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, and J. Vetter. From Interactive Applications to Distributed Laboratories. *IEEE CONCURENCY*, 6, 1999.

[142] P. J Podrid and R. J. Myerburg. Epidemiology and Stratification of Risk for Sudden Cardiac Death. *Clinical Cardiology*, 28:I3–I11, 2005.

[143] A. Promwungkwa. *Data Structure and Error Estimation for an Adaptive p-Version Finite Element Method in 2-D and 3-D Solids*. PhD thesis, The Faculty of the Virginia Polytechnic Institute and State University, 2008.

[144] E. Rank, A. Borrmann, A. Düster, A. Niggl, V. Nübel, R. Romberg, D. Scholz, Ch. van Treeck, and P. Wenisch. From Adaptivity to Computational Steering: The Long Way of Integrating Numerical Simulation into Engineering Design Process. In *Proc. of International Conference on Adaptive Modeling and Simulation, ADMOS*, 2005.

[145] E. Rank, A. Düster, M. Krafczyk, and M. Rücker. Some Aspects of Coupling Structural Models and p-Version Finite Element Methods. *Computational Mechanics. New Trends and Applications*, 1998.

[146] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.

[147] K. Reisdorph. Using Timers. *C++Builder Developer's Journal*, 2000.

[148] K. Rycerz, M. Bubak, P. Sloot, and V. Getov. Problem Solving Environment for Distributed Interactive Applications. In *Proceedings of CoreGRID Integration Workshop*, pages 129–140, 2006.

[149] B. Schauer. Multicore Processors: A Necessity. *Proquest Discovery Guides*, 2008.

[150] J.F. Shepherd and C.R. Johnson. Hexahedral Mesh Generation for Biomedical Models in SCIRun. *Engineering with Computers*, 25(1):97–114, 2009.

[151] J. Shu, Layne T. Watson, Naren Ramakrishnan, and F. A. Kamke. Computational Steering in the Problem Solving Environment WBCSim. *Engrg. Comput.*, 28:888–911, 2011 2011.

[152] B. Simsek. Signals, 2005. http://www.enderunix.org/simsek/.

[153] P. M. A. Sloot, A. Tirado-Ramos, A. G. Hoekstra, and M. Bubak. An Interactive Grid for Non-Invasive Vascular Reconstruction. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 309–319, Washington, DC, USA, 2004. IEEE Computer Society.

[154] M. Steffen, J. Tate, and J. Stinstra. Defibrillation Tutorial, 2012. url: http://www.sci.utah.edu/devbuilds/scirun_docs/DefibrillationTutorial.pdf.

[155] J. Stinstra and D. Swenson. Ischemia Model Tutorial, 2012. url: http://www.sci.utah.edu/devbuilds/scirun_docs/IschemiaModelTutorial.pdf.

[156] K. Stockinger, J. Shalf, K. Wu, and E. W. Bethel. Query-Driven Visualization of Large Data Sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174. IEEE Computer Society Press, 2005.

[157] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci. Interactive Editing of Massive Imagery Made Simple: Turning Atlanta into Atlantis. *ACM Transactions on Graphics (TOG)*, 30, 2011.

[158] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[159] B. Szabó, E. Rank, and A. Duester. *The p-Version of the Finite Element Method*. John Wiley Sons, Chichester, 2004. Chapter 5 in Vol. 1 of Encyclopedia of Computational Mechanics.

[160] B. A. Szabó. Mesh Design for the p-Version of the Finite Element Method. *Computer Methods in Applied Mechanics and Engineering*, 55:181–197, 1986.

[161] T. Teramoto, T. Okada, and S. Kawata. An Education-Support PSE System: TSUNA-TASTE. *Journal of Convergence Information Techonolgy*, 5(4):216–224, 2010.

[162] S. Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra, 1999.

[163] U. Trottenberg, C. W. Oosterlee, and A. Schüler. *Multigrid*. Academic Press, 2001.

[164] T. Tu, H. Yu, J. Bielak, O. Ghattas, J. C. Lopez, K.-L. Ma, D. R. O'Hallaron, L. Ramirez-Guzman, N. Stone, R. Taborda-Rios, and J. Urbanic. Remote Runtime Steering of Integrated Terascale Simulation and Visualization. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06*, New York, NY, USA, 2006. ACM.

[165] R. van Liere and J. J. van Wijk. CSE—a Modular Architecture for Computational Steering. In *Virtual Environments and Scientific Visualization '96.*, pages 257–266. Springer Verlag, Vienna, 1996.

[166] C. Van Treeck, P. Wenisch, A. Borrmann, M. Pfaffinger, M. Egger, and E. Rank. Utilizing High Performance Supercomputing Facilities for Interactive Thermal Comfort Assesment. In *Proceedings of 10th International IBPSA Conference Building Simulation*, Bejing, China, 2007.

[167] J. Vetter and K. Schwan. High Performance Computational Steering of Physical Simulations. In *Proceedings of 11th International Parallel Processing Symposium*, 1997.

[168] J. Vetter and K. Schwan. Techniques for High-Performance Computational Steering. *IEEE CONCURENCY*, 7(4):63–74, 1999.

[169] J. S. Vetter and S. Karsten. Progress: a Toolkit for Interactive Program Steering. Technical report, Georgia Institute of Technology, 1995.

[170] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Washington, DC, USA, 2010. IEEE Computer Society.

[171] S. Wagner, A. Bode, H. Satzger, and M. Brehm. *High Performance Computing in Science and Engineering Garching/Munich 2012*. Bayerische Akademie der Wissenschafen, 2012.

[172] E. Weinan, B. Engquist, X. Li, W. Ren, and E. Vanden-Eijnden. Heterogeneous Multiscale Methods: A Review. *Communications in Computational Physics*, 2(3):367–450, 2007.

[173] D. M. Weinstein, S. Parker, J. Simpson, Zimmerman K., and M. Jones. *Visualization in the SCIRun Problem-Solving Environment*. Elsavier, 2005.

[174] P. Wenisch. *Computational Steering of CFD Simulations on Terflop-Supercomputers*. PhD thesis, Technische Universität München, München, 2008.

[175] P. Wenisch, C. van Treeck, and E. Rank. Interactive Indoor Air Flow Analysis Using High Performance Computing and Virtual Reality Techniques. In *Proceedings of Roomvent*, Coimbra, Portugal, 2004.

[176] H. Wille, E. Rank, and Z. Yosibash. Prediction of the Mechanical Response of the Femur with Uncertain Elastic Properties. *Journal of Biomechanics*, 45:1140–1148, 2012.

[177] Z. Yang. *The Finite Cell Method for Geometry-Based Structural Simulation*. PhD thesis, TU München, 2011.

[178] Z. Yang, C. Dick, A. Düster, M. Ruess, R. Westermann, and E. Rank. Finite Cell Method with Fast Integration—An Efficient and Accurate Analysis Method for CT/MRI Derived Models. In *Proceedings of European Conference on Computational Mechanics—ECCM*, 2010.

[179] Z. Yang, S. Kollmannsberger, A. Düster, M. Ruess, E. Garcia, R. Burgkart, and E. Rank. Non-Standard Bone Simulation: Interactive Numerical Analysis by Computational Steering. *Computing and Visualization in Science*, 14:207–216, 2011. 10.1007/s00791-012-0175-y.

[180] I. Yavneh. Why Multigrid Methods are So Efficient. *Computing in Science and Engineering*, 2006.

[181] Z. Yosibash. p-FEMs in Biomechanics: Bones and Arteries. *Computer Methods in Applied Mechanics and Engineering*, 249–252:169–184, 2012.

[182] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method, Fourth Edition, Vol. 1 Basic Formulation and Linear Programs.* McGraw-Hill Book Company Europe, 1994.